Monterey Workshop 2001

Sponsored by:

ONR / AFOSR / ARO / DARPA

Engineering Automation for Software Intensive System Integration

June 18-22, 2001

U.S. Naval Postgraduate School
Monterey, California

Proceedings of

Monterey Workshop 2001

Engineering Automation for Software Intensive System Integration

Sponsored by:

Office of Naval Research
Air Force Office of Scientific Research
Army Research Office
Defense Advanced Research Projects Agency

June 18-22, 2001
U.S. Naval Postgraduate School
Monterey, California

Workshop Chairs
Luqi, US Naval Postgraduate School
Manfred Broy, Technical University of Munich

Workshop Chairs

Luqi, US Naval Postgraduate School

Manfred Broy, Technical University of Munich

Program Committee

Egidio Astesiano – University of Genova

Mikhail Auguston – New Mexico State University

Valdis Berzins – US Naval Postgraduate School

Swapan Bhattacharya – IIIT, Calcutta

Barrett Bryant – University of Alabama

Jun Ge – US National Research Council

Jiang Guo – US National Research Council

David Hislop – US Army Research Office

Robert Herklotz – US AFOSR

Purush Iyer – Uppsala University

Oleg Kiselyov – US National Research Council

Fabrice Kordon – LIP6-SRC, Universite Paris 6

Zohar Manna – Stanford University

Dave Robertson – University of Edinburgh

John Zavada – ARO/ERO, US Army

Doug Gage - DARPA

Ralph Wachter - ONR

Du Zhang - California State University

Proceedings Editor

Nabendu Chaki - US Naval Postgraduate School

Swapan Bhattacharya - IIIT, Calcutta

Local Arrangements

Mantak Shing Jesse Betts Judy Cabana John Cristobal

Adriane Fells Oleg Kiselyov Ben Quismundo Ryan Yokogawa

List of Attendees in Monterey Workshop 2001

Noel A. Acevedo	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Dagohoy Anunciado	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Egidio Astesiano	Universita' Digenova, Genova, Italy
Mikhail Auguston	Naval Postgraduate School, Monterey, CA, USA
William Bail	The MITRE Corporation and PEO TSC, McLean, VA, USA
Farokh Bastani	University of Texas at Dallas, TX, USA
Daniel M. Berry	University of Waterloo, Canada
Valdis Berzins	U.S. Naval Postgraduate School, Monterey, CA, USA
Swapan Bhattacharya	Indian Institute of Information Technology, Calcutta, India
Pam Binns	Honeywell Laboratories, Minneapolis, MN, USA
John Bohn	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Manfred Broy	Institut Fur Informatik, Technische Universitat Munchen, Germany
Barrett Bryant	U.S. Naval Postgraduate School, Monterey, CA, USA
Carol Burt	2AB Inc, 1700 Highway 31, Calera, AL 35040, USA
Vineet Chadha	Mississippi State University, Starkville, MS, USA
Nabendu Chaki	U.S. Naval Postgraduate School, Monterey, CA, USA
Samiran Chattopadhyay	Jadavpur University, Calcutta-32, India
Andrew Chen	Naval Air Warfare Center, Point Mugu, CA, USA
Ron Chen	Defense Management Data Center, Seaside, CA, USA
Maxwell Chi	Joint Interoperability Test Command, Ft. Huachuca AZ, USA
Lori Clarke	University of Massachusetts, MA, USA
John Clements	Rice University, Houston TX, USA
Robert Cook	Georgia Southern University, Georgia, USA
Dan Cooke	Texas Tech University, TX, USA
Steve Cross	Software Engineering Institute, Carnegie Mellon University, PA, USA
Zhang Cui	California State University, Sacramento, CA, USA
Michael Dabose	Raytheon Missile Systems, West, Tucson, AZ, USA
David Dampier	Mississippi State University, Starkville ,MS, USA
John Drummond	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Kathi Fisler	Worcester Polytechnic Institute, MA, USA
Leonard Gaines	Naval Supply Systems Command, Mechanicsburg, PA, USA

Ann Gates	University of Texas at El Paso, TX, USA
Jun Ge	U.S. Naval Postgraduate School, Monterey, CA, USA
Kevin Greaney	Ballistic Missile Defense Organization, Pentagon, Washington DC, USA
Cordell Green	Kestrel Institute, Palo Alto, CA, USA
Armando Haeberer	Alameda Antonio Sergio 7 Sala 1A 2795, Linda-a-Velha, Portugal
David Hislop	U.S. Army Research Office, NC, USA
Purush Iyer	North Carolina State University, NC, USA
Grant Jacoby	3804 SandTrap Circle, Mason, OH, USA
Craig Johnson	Defense Contract Management Agency, Sunnyvale, CA, USA
Paul Jones	U.S. Food and Drug Administration, Rockville, MD 20857
Dan Kedziorek	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Oleg Kiselyov	U.S. Naval Postgraduate School, Monterey, CA, USA
Fabrice Kordon	LIP6-SRC, Universite Paris 6, Paris, France
Bernd Kraemer	Fern Universitaet, Hagen, Germany
Shriram Krishnamurthi	Brown University, Providence, RI, USA
Bill Lafond	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Doug Lange	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Insup Lee	University of Pennsylvania, Philadelphia, PA, USA
Bruce Lewis	U.S. Army Aviation & Missile Command, Redstone Arsenal, AL, USA
Matthew Lisowski	Naval Strike Air Warfare Center, NV, USA
Nelson Ludlow	Mobilisa, Port Townsend WA, USA
Luqi	U.S. Naval Postgraduate School, Monterey, CA, USA
Milton Mata	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Duane Matlen	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
John Melear	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Bret Michael	U.S. Naval Postgraduate School, Monterey, CA, USA
Christopher Miles	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Ann Miller	University of Missour-Rolla, Rolla, MO, USA
Michael Mislove	Tulane University, New Orleans, LA, USA
Theng Moua	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Michael Murrah	U.S. Naval Postgraduate School, Monterey, CA, USA
Christopher Mushenski	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Gleb Naumovich	Polytechnic University, Brooklyn, NY, USA

Paul Nelson	Standard Army Management Information Systems, Fort Belvoir, VA, USA
Danh Nguyen	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Phuong Phan	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Rob Piirainen	PO BOX 58123 (MD X70), Santa Clara, CA, 95052
Joseph Puett	U.S. Naval Postgraduate School, Monterey, CA, USA
Rajeev Raje	Indiana University Purdue University Indianapolis, IN, USA
Rakhee Ramgolam	U.S. Naval Postgraduate School, Monterey, CA, USA
William Ray	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Joy Reed	Oxford Brookes University, Headington, Oxford, England
Dan Regep	LIP6-SRC, Universite Paris 6, Paris, France
Giana Reggio	Universita' Digenova, Genova, Italy
Richard Riehle	U.S. Naval Postgraduate School, Monterey, CA, USA
Michael Saboe	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Roberto Sandoval	Joint Information Operations Center, San Antonio, TX, USA
Sol Shatz	University of Illinois at Chicago, Highland Park, IL, USA
Lydia Shen	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Mantak Shing	U.S. Naval Postgraduate School, Monterey, CA, USA
Keith Shockley	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Henny Sipma	Stanford University, Stanford, CA, USA
Doug Smith	Kestrel Institute, Palo Alto, CA, USA
Stephen Smith	U.S. Joint Forces Program Office, San Diego, CA, USA
William Smuda	U.S. Army Tank-Automotive & Ammunition Command, Warren, MI, USA
Oleg Sokolsky	University of Pennsylvania, PA, USA
Eugene Stark	State University of New York at Stony Brook, NY, USA
Xian-He Sun	Illinois Institute of Tech, Chicago, IL, USA
Geoffrey Thome	U.S. Naval Postgraduate School, Monterey, CA, USA
Paul Tobin	Armed Forces Communications & Electronics Association, Fairfax, VA, USA
Stephen Vestal	Honeywell Laboratories, Minneapolis, MN, USA
Jennifer Warwick	U.S. Space & Naval Warfare Systems Center, San Diego, CA, USA
Martin Wirsing	Institut fur Informatik - University of Munich (LMU), Germany
Michael Yee	Defense Management Data Center, Seaside, CA, USA
Paul Young	U.S. Naval Postgraduate School, Monterey, CA, USA
Du Zhang	California State University, Sacramento, CA, USA

Preface

Luqi

The 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration was sponsored by the Office of Naval Research, Air Force Office of Scientific Research, Army Research Office and the Defense Advance Research Projects Agency. It is our pleasure to thank the workshop advisory and sponsors for their vision of a principled engineering solution for software and for their many-year tireless effort in supporting a series of workshops to bring everyone together.

This workshop is the 8th in a series of International workshops. The workshop was held in Monterey Beach Hotel, Monterey, California during June 18-22, 2001. The general theme of the workshop has been to present and discuss research works that aims at increasing the practical impact of formal methods for software and systems engineering. The particular focus of this workshop was "Engineering Automation for Software Intensive System Integration". Previous workshops have been focused on issues including, "Real-time & Concurrent Systems", "Software Merging and Slicing", "Software Evolution", "Software Architecture", "Requirements Targeting Software" and "Modeling Software System Structures in a fastly moving scenario".

A major goal for this series of workshops is to encourage the software engineering community in general to improve interaction between researchers and engineering practitioners. The workshop has long established itself as a summit where researchers from academics and industries can exchange recent results, assess their significance and earn motivation for transferring the relevant results to practice. This indeed is a forum where software engineers may communicate current problems in engineering practice to researchers and help focus to bridge the gap between the theoretical and practical sides of the subject.

It is no longer the case that theoretical foundations for computing are lacking. However, keeping in mind the challenge to put these results to work, the formal aspects of computing cannot be studied in isolation in the context of software engineering. The need to ensure that the assumptions on which formal models are based are consistent with the situations encountered in practical applications puts interdisciplinary requirements on researchers and lends importance to interactions between experts from heterogeneous backgrounds.

This year, apart from the distinguished panel of invited speakers, we have accepted contributed papers mainly to encourage the emerging researchers in software engineering. This has widened the scope of discussion and the sessions were highly interactive and rich with intellectual frictions in opinion from a broad range of experts. Members of the academic, government, military and commercial world exchanged their vision, insight and concerns on many important issues. I hope that the workshop has made another step to reduce the gap between theory and practice of software engineering.

Content

Little Languages and their Programming Environments

John Clements[1] Paul Graunke[1] Shriram Krishnamurthi[2]
Matthias Felleisen[1]

[1]Department of Computer Science
Rice University
Houston, TX 77005-1892

[2]Computer Science Department
Brown University
Providence, RI 02912

contact: <sk@cs.brown.edu>

March 15, 2001

Summary

Programmers constantly design, implement, and program in little languages. Two different approaches to the implementation of little languages have evolved. One emphasizes the design of little languages from scratch, using conventional technology to implement interpreters and compilers. The other advances the idea of extending a general-purpose host language; that is, the little language shares the host language's features (variables, data, loops, functions) where possible; its interpreters and compilers; and even its type soundness theorem. The second approach is often called a language *embedding*.

This paper directs the attention of little language designers to a badly neglected area: the programming environments of little languages. We argue that an embedded little language should inherit not only the host language's syntactic and semantic structure, but also its programming environment.

We illustrate the idea with our DrScheme programming environment and S-XML, a little transformation language for XML trees. DrScheme provides a host of tools for Scheme: a syntax analysis tool, a static debugger, an algebraic stepper, a portable plugin system, and an interactive evaluator. S-XML supports the definition of XML languages using a simple form of schemas, the convenient creation of XML data, and the definition of XML transformations.

The S-XML embedding consists of two parts: a library of functions and a set of syntactic extensions. The elaboration of a syntactic extension into core Scheme preserves the information necessary to report the results of an analysis or of a program evaluation at the source level. As a result, all of DrScheme's tools are naturally extended to the embedded language. The process of embedding the S-XML language into Scheme directly creates a full-fledged S-XML environment.

We believe that this method of language implementation may be generalized to other languages and other environments, and represents a substantial improvement upon current practice.

1 Reusing Language Technology

Programmers constantly design little programming languages. Many of these languages die a quick death or disappear under many layers of software; network protocols, GUI layout declarations, and scripting tools are examples. Others evolve and survive to fill a niche; AWK, Make, Perl, and Tcl come to mind.

Once a programmer understands that some problem is best solved by designing a new little language, he must make an implementation choice. One possibility is to build the little language from scratch. This option involves the tasks of specifying a (typically formal) syntax, a (semi-formal) system of context-sensitive constraints, and an (informal) semantics; and of implementing the required software: a lexer, a parser, a type checker, a code generator and/or an evaluator.

The other option is to extend an existing general-purpose language with just those constructs that the task requires. In this case, the little language shares the host language's syntax (variables, data, loops, functions) where possible; its interpreters and compilers; and even its type soundness theorem. This kind of extension is often called a *language embedding.*

The following table summarizes the salient differences between strategy of implementing a language "from scratch" in a language *A* and the strategy of embedding a little language into a language *B*.

designing a little language "from scratch"	embedding a little language
variables, loops, etc. are designed explicitly	variables, loops, etc. are those of **B**
safety/type-soundness may not exist	safety/type-soundness is that of **B**
lexer is implemented in **A**	the lexer is an extension of **B**'s
parser is implemented in **A**	the parser is an extension of **B**'s
validity checker is implemented in **A**	the validity checker is **B**'s
interpreter is implemented in **A**	the interpreter is **B**'s

Succinctly put, the "implement from scratch" strategy *uses* technologies; an embedding shares, and thus truly *reuses*, technology for the construction of a little language.

This paper illustrates that a language embedding can reuse more of the host's technology than just the evaluator. Specifically, we argue that if a programming environment for a host language is properly constructed and if we use a well-designed embedding technology, the mere act of constructing the embedding also creates a full-fledged programming environment for the little languages.

In support of our argument we construct an embedded little language, called S-XML, and derive its environment from DrScheme, our Scheme programming environment [7]. S-XML permits programmers to create and manipulate XML-like data. More precisely, they can use a set of constructs to specify XML trees in a natural manner, and they can define tree transformations on the data with an easy-to-use pattern-matching construct. DrScheme provides a host of tools for Scheme: a syntax analysis tool that includes a variable binding display and a variable renaming mechanism; a static debugger; an algebraic stepper; a portable library system; and an evaluator that correlates run-time exceptions with the program source. the S-XML programming environment inherits all of these.

The S-XML embedding consists of several language extensions. Some can be defined as functions, some cannot. The implementation of the latter exploits DrScheme's syntax definition mechanism, which, in turn, is based on Scheme's macro technology. DrScheme's syntax extensions are completely transparent to DrScheme's tools. At the same time, the elaboration of a syntactic extension into core Scheme preserves all necessary information to report the results of an analysis or of

```
<article>
  <header>
    <title>Not an Article</title>
    <author> John Clements </author>
  </header>
  <text>
    This is not a newspaper article.
    But if it was, this is where the
    article's text would be.
  </text>
</article>
```

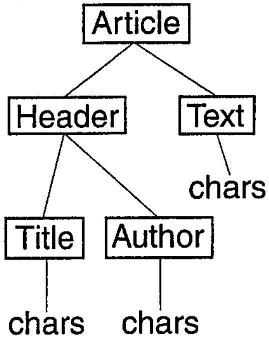


Figure 1: Correspondence between concrete and abstract syntaxes

a program evaluation at the source level. By adding two small extensions that undo the elaboration at certain strategic places, we thus ensure that DrScheme's syntax checker checks the syntax and context-sensitive properties of S-XML transformations; the static debugger turns into an XML validity checker; the stepper shows how the transformations rewrite XML trees at the level of XML data constructors; and the interpreter prints XML results and reports errors in terms of S-XML transformations. In short, the process of embedding the S-XML language into Scheme directly creates a full-fledged S-XML environment.

The following section introduces XML and S-XML; the third section discusses the S-XML embedding in Scheme. The fourth and fifth section present DrScheme and the little language environment created with the embedding. The underlying technology is explained in the sixth section. The seventh section relates our work to the relevant areas. The last section summarizes our ideas and suggest topics for future extensions.

2 A Running Example: S-XML

To Illustrate our ideas, we develop a little language — and an accompanying programming environment — for operating on XML documents.

2.1 XML

XML (for "eXtensible Markup Language") is a proposed standard for a family of languages. It was designed to provide a middle ground between the universally accepted but inconsistent and semantically rigid HTML language and the extensible but overly complex SGML family of languages. To a first approximation, an XML element may be either character data or a tag pair annotated with an optional attribute association list and enclosing a list of zero or more XML elements [3]. In this regard, HTML and XML are similar.

On a deeper level, XML consists of two related parts: a concrete syntax and an abstract syntax. Figure 1 shows an example of the concrete syntax and a corresponding abstract syntax tree.

Specific languages within the XML domain are specified using "schemas". A schema defines the set of valid tags, their possible attributes, and constraints upon the XML elements appearing between a pair of tags. A schema for the newspaper article language from figure 1 appears in figure

```
<schema>
  <element name="header">
    <sequence> <element-ref name="title"/>
               <element-ref name="author"/>
    </sequence>
  </element>
  <element name="body">
    <mixed> <pcdata/> <mixed/>
  </element>
  <element name="article">
    <sequence> <element-ref name="header"/>
               <element-ref name="body"/>
    </sequence>
  </element>
</schema>
```

Figure 2: A simple schema for newspaper articles

2.[1] This schema specifies, among other things, that the *header* field must contain a *title* and an @author@. The ability to specify XML languages explicitly using schemas is what most clearly separates XML and HTML.

XML documents are data; in order to use this data, programmers must write programs that accept and manipulate it. Walsh [27], a member of the XML design team, states:

> ... [I]t ought to take about two weeks for a competent computer science graduate student to build a program that can process XML documents.

The implication is that processing XML data is a tedious and time-consuming process, involving the design and implementation of a project-specific package of I/O routines.

Below the surface syntax, XML expressions are purely trees. Each node is either character data or a tagged node containing a set of attributes and a set of subtrees. A program that processes XML data will be a tree-processing program. Given the complexity of the defined syntax, it makes sense to abstract away from that concrete syntax into a purely tree-based paradigm.

Once the work of parsing concrete syntax is moved out of the programmer's domain, processing XML trees becomes a more manageable task. Many if not most XML programs will consist of a small set of tree transformations, taking the data from one XML language into another. For instance, a newspaper's web site might be designed to transform an article stored in an XML-structured database (as shown in figure 1) into a web page shown to a reader. An HTML document produced by such a transformation is shown in figure 3.

2.2 S-XML

The simple and specialized nature of XML transformations makes them an ideal candidate for an embedded language solution. The language should include special forms for creating and validat-

[1] The W3C has not yet settled on a schema standard. The schema shown here is written in a simple illustrative schema language designed to be read easily. Also, the trivial schemas for *author* and *title* are omitted.

```
<html>
  <head><title>Not an Article</title></head>
  <body>
    <center><h1>Not an Article</h1>by John Clements</center>
    <spacer type="vertical" size="20">
    <p>This is not a newspaper article. But if it was, this
    is where the article's text would be.</p>
  </body>
</html>
```

Figure 3: The result of a simple XML transformation

ing XML elements, and a mechanism for expressing tree transformations easily. On the other hand, a language for XML processing should not preclude the production of more complex programs. Rather, it should allow programmers to work with the full power of the general-purpose host language, if they so choose.

We call our language S-XML. It uses S-expressions to match the tree-based structure of XML elements. It provides the **xml** and **lmx** forms for creating XML elements and embedding computation; the **xml-match** form to state pattern-based transformations on these elements; and a language of schemas to express language restrictions. We explain these constructs below.

2.2.1 xml

The little language must provide language forms for constructing XML elements conveniently, because any program that transforms XML data needs to construct XML elements. In other words, we must choose a concrete syntax for these elements in the embedded language.

To take a simple example, a HTML footer might contain a horizontal line and a page number. A naïve approach would be to directly embed XML's concrete syntax into Scheme strings:

"<center>page number <em>3</em></center>"

The obvious shortcoming of the string representation is its lack of structure; every procedure that operates on this data must parse the string all over again. This is wasteful and time-consuming. A better way is to specify this data in a structured form. Our language should provide a straightforward way to create such "parsed" structures, independent of the representation of these data. Ideally, the program text that creates an XML element should closely resemble the XML text itself, less the end tag. In the S-XML language, this datum is therefore represented with the following program text:

(**xml** (*center* "page number " (*em* 3)))

Within the form (**xml** ...), each nested subexpression is taken to describe an XML element. Just as double-quotes and backslashes are used in many languages to denote literal data, **xml** is used to denote XML literals.

XML elements may also contain attributes. The **xml** form permits the addition of attributes to elements. These attributes appear as an optional (parenthesized) list immediately following the tag name. Thus, an HTML *body* tag with the *bgcolor* attribute might be written as:

```
(define (format-article xml-article)
  (xml-match xml-article (title-string author-string body-text T)      ; keywords
    [(article (header (title title-string) (author author-string))      ; pattern
              (text body-text ...))
     (xml (html (head (title title-string))                             ; result
                (body (center (h1 title-string) "by " author-string)
                      (spacer ((type "vertical") (size "20")))
                      body-text ...)))]
    [(page _)                                                           ; pattern
     (error 'format-page "badly formatted xml-article")]))              ; result
```

Figure 4: A simple transformer

```
(xml (body ((bgcolor "BLUE")) ...))
```

2.2.2 lmx

With the **xml** construct, programmers can conveniently specify large XML constants. But programmers may also wish to abstract such tree constructions over certain parameters. For example, a programmer may wish to specify the footer of a page relative to a page number. To allow an "escape" into the parent language, S-XML includes the **lmx** construct:

```
(lmx expression)
```

An **lmx** expression may only occur as a sub-expression of some **xml** expression. It evaluates its subexpression; the result is spliced into the XML tree in place of the **lmx**-expression. Using a combination of **lmx** and **xml** forms, a programmer can now easily define a function that produces a page footer:

```
(define (make-footer page-number)
  (xml (center "page number: " (em (lmx page-number)))))
```

2.2.3 xml-match

The programmer now has the tools needed to build elements of the desired XML language. Next, he needs a mechanism to manipulate these elements in a simple way. The most convenient method is to use pattern-matching; our S-XML language provides the **xml-match** form, to perform pattern-matching and tree-processing on XML elements.

To evaluate an **xml-match** expression, each pattern is matched against the input. Once a match is found, the result expression is evaluated, with the bindings introduced by the pattern-match.

Figure 4 shows the definition of the HTML-producing transformer illustrated earlier. Note that both input and output patterns are specified in the same way that **xml** elements are.

```
(schema
 (element ((name "header"))
          (sequence (element-ref ((name "title")))
                    (element-ref ((name "author")))))
 (element ((name "body"))
          (mixed (pcdata)))
 (element ((name "article"))
          (sequence (element-ref ((name "header")))
                    (element-ref ((name "body"))))))
```

Figure 5: A S-XML Schema for an Article Language

2.2.4 schema

One of the most important features of XML is the ability to define and restrict XML languages, using formal specifications. Several standards have been proposed for this; S-XML uses our version of *schemas*. A schema describes the set of valid XML elements for a specific XML language. A schema is also itself an XML element, and may therefore be described using the same S-XML conventions. Figure 5 shows the S-XML representation of the schema shown in figure 2. A comparison with the XML specification of this schema reveals the similarity between the two.

3 Building a Little Language

On the one hand, much of the functionality of a little language may be established by building a library of functions and constants. In fact, for some tasks a domain-specific library serves as a complete solution to the embedding problem.

On the other hand, there are language forms that cannot be implemented as ordinary functions. Among these are shortcuts for creating structured data (e.g. **xml** and **lmx**), language forms that introduce variable bindings (e.g. **xml-match**), and language forms that affect the flow of control in non-standard ways (**xml-match** again).

These new language forms may be added using *macros*. Macros are tree-rewriting rules that are applied to syntax trees during compilation. They elaborate the language forms of the little language into the forms of the host language. In our case, the host language is Scheme.

3.1 Scheme Macros

The notion of syntactic abstraction is not a new one. Nearly every general-purpose programming language has some facility for declaring and invoking macros. However, the vast majority of these are deeply flawed. Macro systems like C's gained a well-deserved reputation as dangerous and inelegant. Their ill-considered use often leads to problems for novices and experts alike. Embedding a little language in C using these macros would be difficult at best.

Fortunately, languages like Scheme offer more controlled and useful macro mechanisms. These systems operate on expressions, rather than tokens, and they have a well-defined semantics as tree transformations. As a simple example, consider the **let** form of Scheme. The **let** form binds values to variable names. In many languages, this type of operation is built into the language. In Scheme, it need not be. Instead, Scheme may implement **let** with a macro that elaborates each use of the form into the application of a procedure. Here is the rewriting rule for **let**:

(**let** ((<*var*> <*exp*>) ...) <*body*> ...)) ↦ ((**lambda** (<*var*> ...) <*body*> ...) <*exp*> ...)

The ellipses are not a notational shorthand but are an integral part of the macro language described in the Revised^5 Report on Scheme [14]. On the left-hand-side of the macro, they indicate that the prior pattern will occur zero or more times, as in a BNF grammar. This input pattern is matched against the input, and where ellipses occur, bindings of lists are created. The right-hand-side pattern uses ellipses to generate sequences of output patterns drawn from these bindings. The components of the matched patterns may be split from each other, as illustrated by the **let** macro shown here.

3.2 Building S-XML

S-XML is implemented as an embedding within Scheme. The embedding (comprising the forms enumerated in section 2.2) is constructed as a combination of a small functional library and a set of macros.

The **xml** form is implemented as a single macro. This macro transforms uses of the **xml** form into expressions that construct Scheme data. The form also permits the omission of empty attribute fields; it is this kind of syntactic shorthand that gives the little language one of its true advantages over the unmodified general-purpose language. The action of the **xml** macro is shown in this example, where an **xml** form is translated into Scheme code that creates a structure:

(**xml** (*center* "Text: " (**lmx** (*get-text*)))) ↦ (*make-center* (*list*) (*list* "Text: " (*get-text*)))

Each use of the **schema** form elaborates into a structure declaration and a type declaration.[2] An example of this macro's translation is shown here:

```
(schema                                           (begin
  (element ((name "elt"))                    ↦      (define-struct elt (attrs elements))
    (sequence                                       (define-type elt (cons other null)))
      (element-ref ((name "other"))))))
```

Note that adopting a richer schema language is simply a matter of modifying a single macro; no other code needs to change.

The **xml-match** form is implemented using a macro in conjunction with a library function. The macro delays the evaluation of the patterns and their matching expressions. It also provides bindings for any pattern variables that occur in the expressions. The function accepts a value and these pattern-expression pairs, and evaluates the first expression whose pattern matches the input value.

[2] DrScheme uses a type inference system called MrSpidey, described in more detail in section 4.

A transformer that takes centered text to italicized text is elaborated like this:

```
                                 (xml-match-fn (xml (center 3))
(xml-match (xml (center 3))        (list 'text)
  (text)                     ↦     (list
  ((xml (center text))               (list '(center text)
   (xml (italic text))))                   (lambda (text)
                                             (xml (italic text))))))
```

The *xml-match-fn* procedure is a part of S-XML's runtime library.

With the addition of these three forms, Scheme becomes S-XML, a little language ideal for constructing and manipulating XML-like data, along with the full gamut of Scheme values. Variables and functions are inherited from Scheme. As a result, first-semester undergraduates can program using XML in a matter of days, rather than the weeks of work that are supposedly required.

4 DrScheme

Building an S-XML evaluator using macros and functions is not enough. This is the lesson that we as programmers have learned in the course of implementing many languages, both little and large. In fact, for a "from scratch" little language implementation, the execution framework is a small fraction of the total work required to make the language usable. To use a language productively, programmers need a host of related tools: editors, checkers (syntax and semantic), debuggers, libraries, and the like. We demonstrate these ideas with the DrScheme programming environment [7].

DrScheme is a programming environment for the Scheme language. It is a graphical, cross-platform environment for developing programs. It includes a syntax-sensitive editor, a read-eval-print loop, a syntax checker, a stepper, and a static type checker. The challenge is to reuse these tools in the design and execution of an embedded language.

Scheme programs are composed entirely of S-expressions, and DrScheme's editor takes advantage of this in many ways. It provides a set of S-expression-directed movement and editing functions. It supports dynamic parenthesis-matching, as well as static highlighting of S-expressions adjacent to the cursor. DrScheme automatically indents lines, and unmatched parentheses are highlighted in red.

Another of the tools DrScheme provides is a syntax-checker. This tool performs a number of tasks:

1. it identifies and highlights syntax errors;
2. it highlights unbound identifiers;
3. it draws arrows from bound identifiers to their binding occurrences; and
4. it permits alpha-renaming, whereby all occurrences of an identifier in a given declaration scope may be renamed consistently.

The syntax checker is useful for beginners, as it helps them to understand the syntax of the source language. The checker is also useful for experienced programmers, who generally make more syntactic mistakes than they would like to admit.

DrScheme also features a symbolic algebraic stepper, which can display a program's execution as an algebraic calculation, according to a standard reduction semantics for Scheme. The stepper shows each step of the execution as a rewriting step; the "before" and "after" expressions are displayed, and the difference is highlighted. The stepper is useful both in debugging and in understanding the details of the language semantics.

DrScheme provides static type-checking through MrSpidey [8]. MrSpidey performs type inference by using set-based analysis [10, 2] to associate a set of values with each program location. When MrSpidey cannot guarantee that the application of a primitive will not cause an error, it flags the location of the primitive's application. Furthermore, MrSpidey provides useful information to the user in the form of graphical inference chains. If an inappropriate argument might reach a primitive, MrSpidey visually depicts the execution path whereby this argument arrives at the erroneous application.

MrSpidey also has an explicit assertion mechanism, of the form (: *expression type*). Using this form, the user may force MrSpidey to check whether an expression is guaranteed to evaluate to a given type. So, for instance, the assertion (: (+ 3 5) *str*) fails, because the result of evaluating (+ 3 5) is a number rather than a string.

DrScheme supports plugins, called Teachpacks for historical reasons. Any DrScheme program may be evaluated with one or more plugins enabled. These plugins are encapsulated using DrScheme's unit system [9], which guarantees that only the intended plugin's functions are exposed, and also that the plugin's meaning will not be affected by the user's code.

5 Building a Little Language Environment

In order to deliver a useful programming environment to the programmer, DrScheme's tools must work seamlessly with the new forms of S-XML. In the following sections, we examine several of DrScheme's tools and how their behavior must change to accommodate the embedded language.

5.1 Editing

Since the little language consists entirely of tree-structured expressions, the editor's features are inherited immediately; editing programs in the little language is as convenient as editing Scheme. The only modification required to the programming environment is the addition of the **xml-match** keyword to the list of specially indented keywords in DrScheme's preference panel.

5.2 Check Syntax

The Check Syntax tool is designed to work transparently through macros. No modification whatsoever is required to extend the syntax checker for an embedded language.

The syntax checker is particularly useful for embedded languages, where the language's syntax is often described informally. For instance, even an experienced programmer might be surprised when using an embedded language to discover that certain identifiers are unbound, or are bound to locations other than expected.

For an example of this, see figure 6, an example using the S-XML language. In particular, this example shows the definition of a simple web page, using the **xml** and **lmx** forms. The bind-

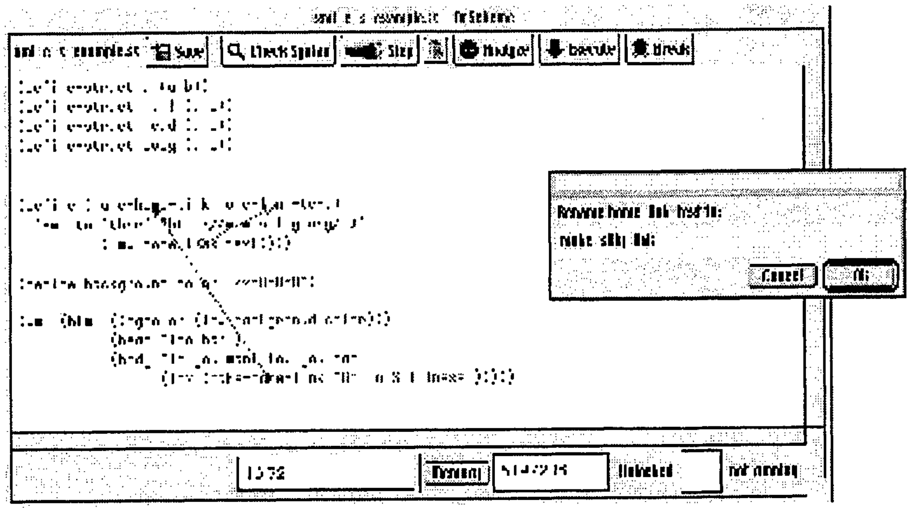


Figure 6: Check Syntax works through macros

ing arrows show how *make-home-link* and *home-link-text* are bound, and the red highlighting[3] on *backgronud-color* indicate that this identifier is unbound (in this case, because of a simple typo). Finally, the 'rename ... to' box shows how users can rename all occurrences of a specific binding in an S-XML transformation.

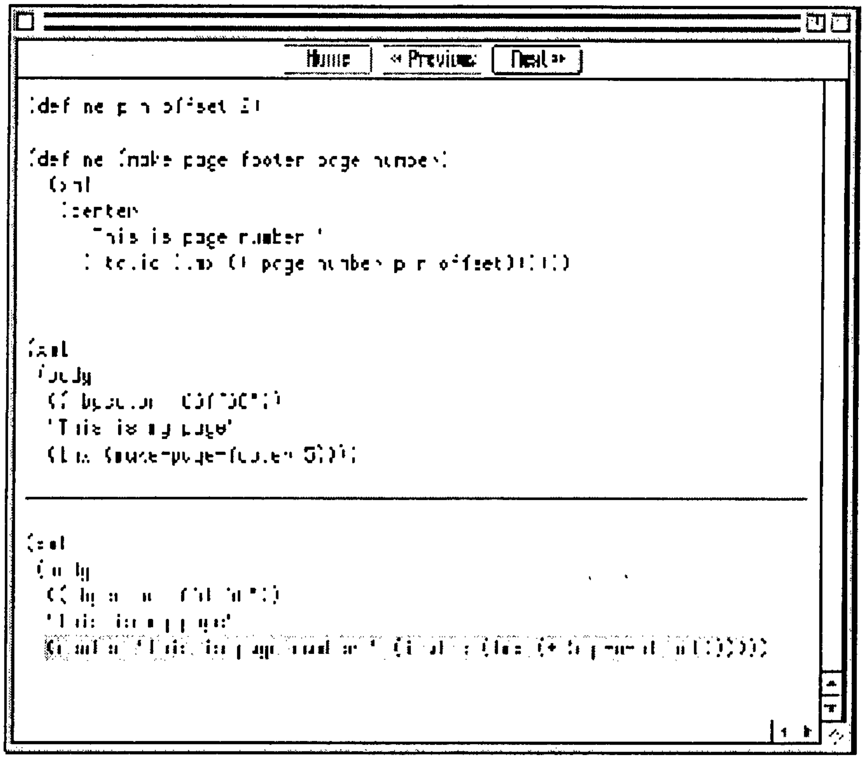
5.3 The Stepper

When a programmer embeds a little language within Scheme, the stepper should be transparent with respect to the macros and libraries introduced by the embedded language. In other words, it must "step" in a manner that corresponds to the reductions of the embedded language, rather than the host language.

S-XML embeds several forms within Scheme; each has a natural reduction sequence. The **xml** form must simply be transparent; **xml** values are displayed as such, and computation within these terms (using the **lmx** form) are properly embedded. The **schema** form is trivial, as it contains no runtime computation. The **xml-match** form shows steps corresponding to the location of the proper pattern, and those within the corresponding pattern.

Figure 7, shows a step in the evaluation of a simple HTML construction. The stepper highlights the reducible subexpression in green, and the resulting subexpression in purple. The call to *make-page-footer* is replaced by the body of the procedure, and the value of the argument is substituted in the bound location in the body.

[3] On a grayscale printer, this will appear gray.



```
(define pn-offset 2)

(define (make-page-footer page-number)
  (xml
   (center
     "This is page number "
     (italic (em (+ page-number pn-offset))))))


(xml
 (body
  ((bgcolor "C0C0C0"))
  "This is my page"
  (hr (make-page-footer 5))))
```

```
(xml
 (body
  ((bgcolor "C0C0C0"))
  "This is my page"
  (center "This is page number " (italic (em (+ 5 pn-offset))))))
```

Figure 7: The stepper works through macros

5.4 Validity Checking

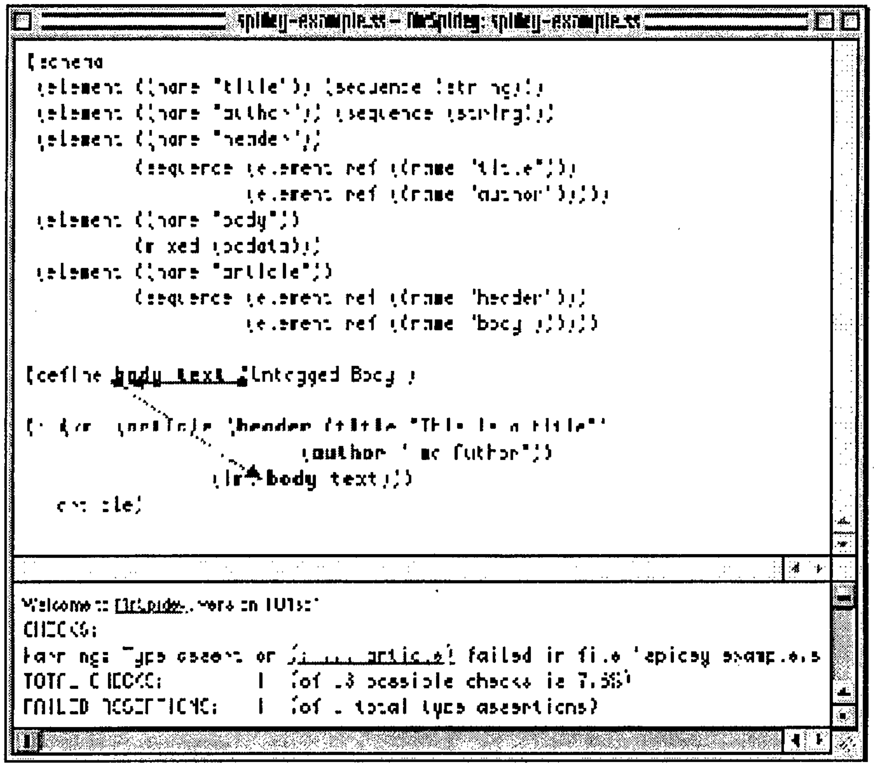
MrSpidey provides an assertion mechanism to enable programmers to check statically that certain variables may only be bound to values of a given type. The natural extension of this assertion ability in the S-XML language is to use the assertion operator for validity checking. In S-XML, a schema expands into a MrSpidey type definition.

This type definition may then be used to implement S-XML validity checking, as shown in figure 8. Rather than a *body*, this *article* has simply a string. This is illegal, by the schema that appears above. Therefore, MrSpidey highlights the offending assertion in red. The path from the string to its use in the **xml** form is indicated by a series of arrows.

5.5 Plugins

DrScheme's plugin system also proves useful in the S-XML language embedding. For instance, the "simple-cgi" plugin permits users to build and test cgi scripts. Using this plugin, programmers can write programs which interact directly with a web browser, either by using a simple "question & answer" interface, or by sending a complete HTML form. The S-XML language provides the needed forms to easily construct these HTML forms.

In figure 9, the simple-cgi Teachpack is used to interact with the user directly. Note that in this



```
(schema
 (element ((name "title")) (sequence (string)))
 (element ((name "author")) (sequence (string)))
 (element ((name "header"))
         (sequence (element-ref ((name "title")))
                   (element-ref ((name "author")))))
 (element ((name "body"))
         (mixed (pcdata)))
 (element ((name "article"))
         (sequence (element-ref ((name "header")))
                   (element-ref ((name "body")))))

(define body-text "Untagged Body")

(: (xml (article (header (title "This is a title")
                         (author "An Author"))
                 (br body-text)))
   article)
```

```
Welcome to MrSpidey, version 101p1
CHECKS:
Warning: Type assertion on (: ... article) failed in file "spidey-example.ss
TOTAL CHECKS:          1  (of 13 possible checks is 7.5%)
FAILED ASSERTIONS:     1  (of 1 total type assertions)
```

Figure 8: MrSpidey catches validity errors

case, we may also use the stepper to trace the execution of the script.

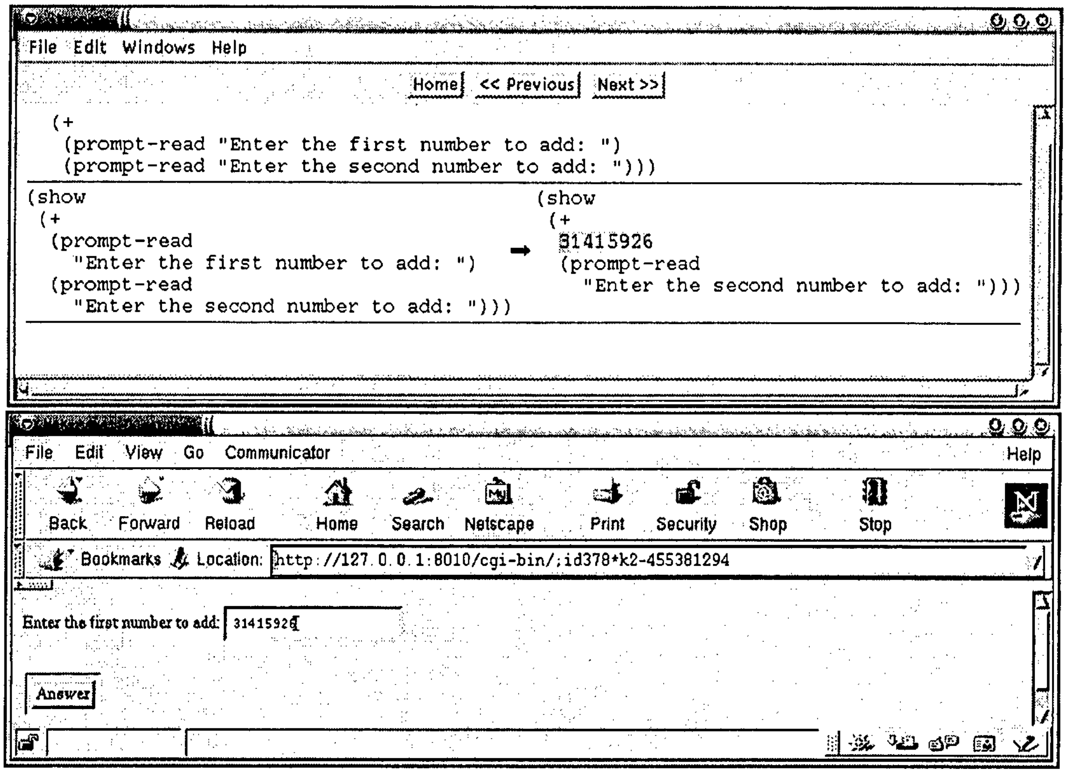
6 How It All Works

The extension of DrScheme's programming tools to S-XML is largely automatic. The key technologies required are source correlation and *rectifiers*.

In DrScheme, source elaboration of macros is performed by McMicMac [19]. McMicMac transforms a source file (a character stream) into an abstract syntax tree. Each term in the tree has a reference to some position of the source file. These references are preserved by McMicMac's subsequent macro elaboration, so that each term in the fully elaborated program has a direct reference to a source location. This elaborated program goes to the evaluator for execution.

As a consequence, the static tools (including the syntax-checker and MrSpidey) operate transparently with respect to macros. These tools draw conclusions about the elaborated program, and display the results using source-correlation indirection. Hence, they require no modification whatsoever to accommodate the embedded language.

The interpreter and the stepper draw heavily on source correlation as well. However, since these tools are not static, they must also display the runtime values and expressions of the embedded language. DrScheme employs rectifiers to perform these back-translations. There are two types of



```
 (+
  (prompt-read "Enter the first number to add: ")
  (prompt-read "Enter the second number to add: ")))
```

```
(show                                     (show
 (+                                        (+
  (prompt-read                        →     31415926
    "Enter the first number to add: ")      (prompt-read
  (prompt-read                                "Enter the second number to add: ")))
    "Enter the second number to add: ")))
```

http://127.0.0.1:8010/cgi-bin/;id378*k2-455381294

Enter the first number to add: 31415926

Figure 9: Using Plugins to Test CGI Scripts

rectifiers: value rectifiers, and expression rectifiers.

A little language that enriches the value set of the host language must include a way to display its values to the user. Value rectifiers perform this translation. That is, if the little language introduces new language forms for the creation of data, the programming tools should display the resulting values using the same forms that the programmer employed to create the data. In S-XML, the following interaction[4] illustrates this:

> (**xml** (*center* "page number " (*em* (**lmx** (+ 1 2)))))
(**xml** (*center* "page number " (*em* 3)))

Rather than displaying the value in an internal format, the printer uses the concrete syntax associated with the little language. Since value rectifiers deal exclusively with runtime values, they have no need of source correlation. A value rectifier provides a mapping from values to displayed information.

The second category of rectifier comprises the expression rectifiers. These arise in the operation of the stepper, which must reconstruct each step within the host language's evaluator as a step within the embedded language. In some cases, the elaborated forms may have been partially evaluated. For instance, the evaluation of the **xml-match** form may proceed through many reductions. Each

[4] Value rectifiers are currently implemented for the stepper, but not for the read-eval-print loop.

of these must be displayed as an **xml-match** term. Expression rectifiers make heavy use of source correlation information, as they must reconstruct source terms based upon the history of macro elaboration imposed upon the source.

For the S-XML language, we have constructed these rectifiers explicitly. Future work includes generating them automatically from the macros and libraries that make up the language embedding.

7 Related Work

Our work relates to four distinct areas of research. They are, in descending order of relevance: the construction of programming environments; the embedding of little languages in host languages; the problem of debugging optimized code; and transformation languages for XML. EMACS is by far the most prominent effort to produce an extensible and customizable programming environment [23]. With a few hundred lines of EMACS code, a programmer can create an EMACS mode that assists with some syntactic problems (indentation, syntax coloring) or with a read-eval-print loop (source correlation of run-time environment). But, the EMACS extensions have to be produced manually; they are not connected or derived from the little language embedding.

Most other work on the construction of programming environments focuses on the creation of tools from language specifications. For example, Teitelbaum, Reps, and others have created the Cornell Synthesizer Generator [21], which permits programmers to use attribute grammar technology to define syntax-directed editors. The ASF+SDF research effort [16] has similar, but more comprehensive goals. A programmer who specifies an algebraic-denotational semantics for a little language can create several interesting tools in this framework. In contrast, our work concentrates on the pragmatic problem of creating or prototyping language tools rapidly. In particular, we accommodate an existing implementation without any modifications. Given that most implementations are not derived formally, our work has greater potential to be applied to other environments.

Second, our most interesting technical problem concerns the relationship between the execution of elaborated code and the source text. At first glance, this suggests a commonality between our work and the work on debugging optimized object code. More specifically, code optimizations are problematic for debuggers and our algebraic stepper. Both need to cope with code transformations when they interrupt the execution of a program. Hennessy [11], Adl-Tabatabai and Gross [1], and Cooper, Kennedy and Torczon [4] describe solutions to the problem of debugging optimized code. We believe, however, that the two communities apply different techniques for the backwards translations due to the radically different levels of languages. We are currently studying whether the techniques carry over from the debugging to the stepping problem and whether the adaptation of these techniques has any advantages.

Third, although our paper is not about techniques for language embeddings, it heavily draws on ideas in that area. The history of language embeddings starts with LISP [24] and McIlroy, who introduced the notion of macro transformations in 1962 [20]. Over the past decade, the Scheme programming language introduced three important innovations in macro systems. First, Kohlbecker, et al. [17] showed how to render macro expanders *hygienic*, that is, make them compatible with the lexical structure of a host language. Second, Kohlbecker and Wand introduced the macros-by-example specification method [18]. Last, but not least, Dybvig, Hieb and Bruggeman [5] implemented the first source-correlating macro system; our work is based on the more powerful McMicMac program elaborator[19].

More recently, other language communities have rediscovered the idea of embedding languages for reuse. Fairbairn [6], Hudak [12], Wallace and Runciman [26] use Haskell's infix operators and higher-order functions to embed little languages,[5] including a little language for XML; Kamin and Harrison [13] are working along similar lines, using SML. More recently, Oleg Kiselyov [15] has also worked to embed XML within Scheme. All of these efforts focus on embedding techniques; none has paid attention to the programming environments of little languages.

Fourth, our paper, like that of Wallace and Runciman [26] and Thiemann[25] address the problem of transforming XML elements. Our solution solves a problem from which both of the other approaches suffer. Specifically, using S-XML programmers can specify XML trees in a generic manner yet they still get the benefits of XML validity checking.

8 Conclusion

We must learn to re-use all levels of language technology in the construction of little languages. The potential benefits are enormous. Shivers [22] reports that his version of AWK, which is more powerful than the original, is one tenth of the original's size. A small implementation is also easy to manage and to change. Hence, an embedded language is easier to extend than a stand-alone language. An improvement to the host language generally improves the embedded language(s) immediately. Finally, if one language plays host to several embedded languages, programs in the latter can easily exchange structured forms of data, e.g., lists, trees, arrays. In contrast, stand-alone implementations must employ the operating system's tool box, which often means that "little language programmers" must write parsers and unparsers.

With this paper we wish to contribute to the argument for language embeddings, and we hope to direct the attention of researchers to the programming environments of little languages. More centrally, we illustrate how an embedding also creates a powerful programming environment for little languages. The construction hinges on three properties of the host language and environment. First, the host language must have a mechanism for defining new language constructs. Otherwise the user of a little language must immediately know everything about the host language. Second, the mechanism must translate instances of the new constructs in such a manner that the tools can report results in terms of the surface syntax. Finally, the tools must not contain hard-wired assumptions about the source language.

For our example, we had to add two small functions to two environment tools: one for translating Scheme values back into S-XML syntax, and another one for reconstructing an S-XML construct that has a multi-step algebraic reduction semantics. Based on our experience, we conjecture that this effort can be automated and we plan to tackle the problem in the future.

References

[1] Adl-Tabatabai, A.-R. and T. Gross. Source-level debugging of scalar optimized code. In *Programming Language Design and Implementation*, May 1996.

[5] These efforts use higher-order functions to express little language programs because the chosen host languages do not provide facilities for defining new language constructs that declare variables. A detailed discussion of this distinction is irrelevant to the topic of our paper.

[2] Aiken, A. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 1999.

[3] Bray, T., J. Paoli and C. Sperberg-McQueen. Extensible markup language XML. Technical report, World Wide Web Consortium, Feburary 1998. Version 1.0.

[4] Cooper, K. D., K. Kennedy, L. Torczon, A. Weingarten and M. Wolcott. Editing and compiling whole programs. In *Software Engineering Symposium on Practical Software Development Environments*, December 1986.

[5] Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[6] Fairbairn, J. Making form follow function: An exercise in functional programming style. *Software—Practice and Experience*, 17(6):379–386, June 1987.

[7] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, number 1292 in Lecture Notes in Computer Science, pages 369–388, 1997.

[8] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[9] Flatt, M. and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[10] Heintze, N. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[11] Hennessy, J. Symbolic debugging of optimized code. *Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[12] Hudak, P. Modular domain specific languages and tools. In *International Conference on Software Reuse*, 1998.

[13] Kamin, S. and D. Hyatt. A special-purpose language for picture-drawing. In *USENIX Conference on Domain-Specific Languages*, 1997.

[14] Kelsey, R., W. Clinger and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.

[15] Kiselyov, O. Scheme and XML. Unpublished Manuscript. Available on the web at: http://pobox.com/ oleg/ftp/Scheme/xml.html.

[16] Klint, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[17] Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

[18] Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[19] Krishnamurthi, S., M. Felleisen and B. F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, September 1999. To appear in Springer-Verlag Lecture Notes in Computer Science.

[20] McIlroy, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, 1960.

[21] Reps, T. W. and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[22] Shivers, O. A universal scripting framework or, Lambda: the ultimate "little language". In Jaffar, J. and R. H. C. Yap, editors, *Concurrency and Parallelism: Programming, Networking and Security*, pages 254–265. Springer-Verlag, 1996. LNCS 1179.

[23] Stallman, R. EMACS: the extensible, customizable, self-documenting display editor. In *Symposium on Text Manipulation*, pages 147–156, 1981.

[24] Steele, G. L., Jr. and R. P. Gabriel. The evolution of Lisp. In Bergin, T. J., Jr. and R. G. Gibson, Jr., editors, *History of Programming Languages—II*, pages 233–308, 1996.

[25] Thiemann, P. Modeling HTML in Haskell. In *Practical Applications of Declarative Languages*, January 2000.

[26] Wallace, M. and C. Runciman. Haskell and XML: Generic document processing combinators vs. type-based translation. In *ACM SIGPLAN International Conference on Functional Programming*, September 1999.

[27] Walsh, N. A technical introduction to XML. *World Wide Web Journal*, Winter 1997.

XML-Based Integration of Interface Definition Language Extensions

Bernd J. Krämer
Department of Electrical and Information Engineering
FernUniversität
58084 Hagen, Germany
bernd.kraemer@fernuni-hagen.de

H.-Arno Jacobsen
Department of Electrical and Computer Engineering and
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
jacobsen@eecg.toronto.edu

1 Introduction

Standard middleware platforms offer interface definition languages (IDLs) to support component reuse and interoperability in a heterogeneous computing context. IDLs typically allow the specification of component and interface names, the signature of operations a component can perform, and possible exceptions that might be raised during operation execution. This limitation to syntactic aspects ensures that IDLs are applicable to a wide range of application domains, can be mapped to a variety of implementation languages, and are easy to learn.

When components are used in mission critical applications, however, more documented information is needed about essential properties of a component to determine how it will behave in the intended context. In [3] Beugnard et al. distinguished four levels of abstraction to organize the specification of a component's properties:

- a *syntactic level*, which is covered, e.g., by OMG IDL, ODL, DCE IDL, or M-IDL,
- a *behavioral level* addressing functional properties of operations, e.g., in terms of pre- and post-conditions,
- a *synchronization level* taking into account that a component might be operating in a concurrent environment, and

- a *quality of service (QoS) level* at which timing requirements, throughput, or precision attributes can be formally documented.

In the literature, we can find a range of proposals for extending IDLs with behavioral specifications [31], synchronization constraints [16], real-time requirements [25, 29], or quality of service specifications [32]. In essence, such extensions break through the abstractions provided by the middleware platform at hand and allow users to specify aspects of an application normally hidden in code and behind transparency mechanisms of the actual middleware.

A crucial issue of all these proposals is, however, the question how they are implemented. A straight-forward attitude is to wait for the proper extension of the proposer's preferred middleware standard and its implementation by a vendor. But standardization endeavors take time and sometimes never materialize. Conversely, applications that would exploit such IDL extensions prior to standardization are not portable as long as they depend on proprietary platform extensions.

To escape this trap, we proposed in [12] to leave IDL untouched and rather include synchronization constraints as comments in IDL interface definitions. The IDL file is then processed as usual with a standard IDL compiler, while the annotated IDL specification is compiled separately with a dedicated tool into code implementing proper sanity checks. In [13] this idea was carried further to the presentation of a collection of design patterns proposing ways to integrate code fragments implementing IDL extensions transparently with the skeleton code generated by standard IDL compilers. The price for this approach was the need for a disciplined use of comments in the IDL specification to avoid that the tool processing the synchronization constraints gets confused by informal comments.

In this paper we further explore this idea by using the eXtensible Markup Language XML [4] as a meta grammar to specify the syntax of the IDL and the given specification extension uniformly. A first design of this idea was published in [14]. The structured XML document provides an adequate basis for a semantic processing of extended component specifications. We sketch the design of a tool that relies on emerging XML technology such as XLS (eXtensible Stylesheet Language) Transformations (XSLT) to process the extended interface definition language. Although originally designed for presentation purposes, XLST can also be used to traverse an XML document and transform it into programming language code, in our case code that implements functional specifications, synchronization constraints, or QoS requirements. This concept builds on the old idea of syntax-directed translation schemes [1].

The remainder of this paper is organized as follows. Section 2 surveys some interface definition language extensions described in the literature. Section 3 develops the framework for specifying IDL extensions and demonstrates how to express IDL with XML and codify IDL extensions with XML. Section 4 sketches the design of a language processing tool based on syntax-directed translation schemes. In the appendix we include the full XML DTD for OMG IDL.

2 Survey on IDL Extensions

IDLs have been designed to provide interfaces to components, while keeping the interface independent of the choice of the actual component implementation language. The design of an IDL is also constrained by the set of intended target languages. For OMG IDL, for instance, this set includes languages as diverse as C, C++, Java, Cobol, Ada, and Lisp.

2.1 IDL Extensions

Much research work has focused on annotating IDL with behavioral extensions, such as pre- and post-conditions, invariants, abstract operation semantics, data integrity conditions, and Horn clauses [24, 23, 6].

Synchronization level specifications state legal partial orderings of operation invocations at a component's interface. They reflect causal dependencies between services provided by a component and give rise to static and dynamic checking of a client-server interaction. Path-expressions [27], IPDL (Interaction Protocol Definition Language) [5], and regular types for active objects [19] were proposed to enforce the sequencing of operation executions, while Petri nets [8] and other mechanisms for expressing richer synchronization constraints are presented in [7, 26, 16]. They address additional properties such as mutual exclusion, synchronization distance, and fairness for protecting shared resources in concurrent environments.

Specification notations and implementation concepts for handling real-time constraints (such priorities, deadlines, or execution time) and QoS attributes (such as allowed response delay, required bandwidth, resource needs, or precision of response) have been proposed in [25, 2, 29, 32].

Other IDL extensions include object co-location constraints and coordination constraints [10], data parallelism [15], security annotations [9], and component definition language extensions [18, 20].

2.2 Processing Extended IDL

In CORBA, DCE, DCOM, and many RPC systems, an IDL specification is processed by a *stub compiler* that generates stub code for client and server side use. This stub code enables communication between components across address spaces and machine boundaries according to the mechanisms of the underlying middleware standard. The stub code manages the packaging and marshaling of service invocations on the client side and provides corresponding reverse operations (unmarshaling, unpackaging) on the server side.

For the implementation of IDL extensions various solutions have been presented in the literature. Many of the above listed language extensions directly modify the IDL syntax by including additional keywords (e.g., [24, 30]). This approach is tied to the particular platform used and the resulting application code is not portable.

In previous work we have proposed to extend IDL by embedding synchronization annotations as comments [16]. This has the advantage that extension unaware IDL compilers are still able to process the specification, whereas extension aware compilers may fully exploit the annotations. The drawback of this approach that an undisciplined provision of comments may confuse the extension aware compiler.

A third alternative relies on programming conventions. Interface attributes and operation arguments are used to encode specifications of real-time constraints as proposed in [25, 29]. The programmer has to obey naming conventions that are enforced by a specialized language processor, which is also responsible for providing the proper stub code. This solution is constrained by the same portability argument as the first approach.

In [13] we have developed the *extension adapter* design pattern that allows application developers to integrate the code implementing his IDL extensions with the server side skeleton code generated from the standard stub compiler. This approach is applicable to all IDL extensions that operate on a component's state across multiple operation invocations.

In contrast to this category of extensions, some of the QoS notations discussed in the literature require support from lower middleware layers. As this support is not available in standard platforms, a portable approach for these extensions is feasible by now.

3 XML Based Framework for Extended IDL

XML has become a popular language for describing structured documents. Although often stated, XML languages have no predefined application-level processing semantics and XML processors have no inherent understanding of document semantics. XML just captures a document's syntactic structure. But a growing number of XML related languages and tools for processing XML specifications are becoming available. Examples include: XSL, the style sheet language; XSLT, the XML transformation language, which uses XSL, and XPath, a language for addressing parts of an XML document.

The integration of XML technology with middleware platforms is an emerging paradigm [28]. Compilers for translating IDL specifications into XML, for instance, exist [22]. Most major database management systems support XML. It is therefore straightforward to create a repository for the extended interface definition language. Most middleware products implement their own repositories, which cannot be extended, to also manage the proposed IDL extensions.

In the sequel we use OMG IDL and the features of the CORBA middleware platform to illustrate our ideas. However, our approach extends to other middleware platforms, such as DCOM (Distributed Component Model) with the M-IDL (Microsoft Interface Definition Language), as well.

3.1 A DTD for OMG IDL

We start from a document type definition (DTD) for IDL, which is compliant to the OMG CORBA standard [21] (see the Appendix for a full reference DTD of OMG IDL). Then we show how to model and integrate IDL extensions in terms of DTDs. Figures 1 and 2 illustrate the tree structure spanned by the IDL constructs `interface` and one of its constituents, namely `operation`. (We avoid presenting the textual form of an XML specification as its readability is limited.)

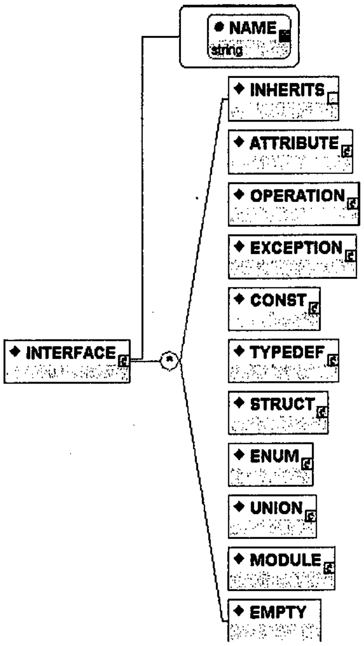


Figure 1: Structure of the DTD for the IDL interface construct

A component interface in IDL has a `name`, a possibly empty list of names of `interfaces`, zero or more `attributes`, `operations` and type signatures, `exceptions`, and few other elements. The details of an `operation` signature are depicted in Fig. 2.

An IDL operation consists of a return type (`typeref`), an operation `name`, an optional qualifier `oneway` that, when available, indicates a non-blocking operation invocation, a parameter list whose elements are qualified as `in`, `out` or `inout` parameters, a clause for raising an exception (`RAISES`), and a `context` clause. Fig. 3 shows an example of an IDL interface including two `operations` and two `attributes`. The corresponding XML code is given in Fig. 4.

Assuming an XML-capable editor, the specification of a component interface in XML is not much different from the specification of the interface in IDL. The XML specification can be easily processed to yield the tag-free interface specifica-

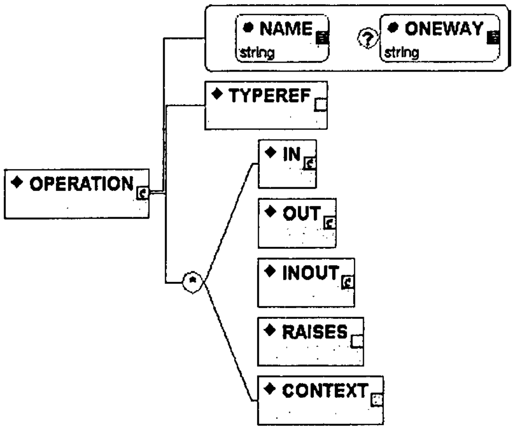


Figure 2: Structure of the DTD for an IDL operation

```
Interface BankAccount {
     void deposit (in Euro amount);
     void withdraw (in Euro amount);
     Euro balance ();
     Euro overdraftLimit (); }
```

Figure 3: IDL specification of the interface of a simple bank account

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE IDL SYSTEM "idl.dtd">
<IDL>
        <INTERFACE NAME = "BankAccount">
                <OPERATION NAME = "deposit">
                        <TYPEREF NAME = "void"/>
                        <IN NAME = "amount">
                                <TYPEREF NAME = "Euro"/>
                        </IN>
                </OPERATION>
                <OPERATION NAME = "withdraw">
                        <TYPEREF NAME = "void"/>
                        <IN NAME = "amount">
                                <TYPEREF NAME = "Euro"/>
                        </IN>
                </OPERATION>
                <ATTRIBUTE NAME = "balance">
                        <TYPEREF NAME = "Euro"/>
                </ATTRIBUTE>
                <ATTRIBUTE NAME = "overdraftLimit">
                        <TYPEREF NAME = "Euro"/>
                </ATTRIBUTE>
        </INTERFACE>
</IDL>
```

Figure 4: IDL specification of the interface BankAccount

tion satisfying the OMG standard.

3.2 Adding a DTD for Synchronization constraints

We specify IDL extensions in the same way as we did with standard IDL but take into account that both DTDs can be merged and the new DTD becomes a new element of the bare IDL DTD (i.e., the merged DTD replaces the "old" IDL DTD). Figure 5 presents the tree representation of an incomplete DTD for specifying synchronization constraints as proposed first in [16]. (Element `other` is just an abbreviation for other constraint operators, which are of no interest here.)

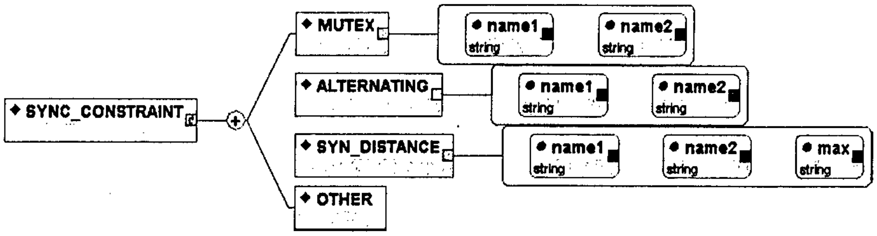


Figure 5: Structure of the DTD for synchronization constraints

Merging this DTD and the IDL DTD introduced in the previous subsection yields a new DTD in which `sync_constraint` is a new alternative in the list of elements of an `interface`. The annotated IDL specification in Fig. 6 illustrates both the inclusion of synchronization constraints and behavior specifications inspired by [3]. The DTD defining the syntax of the behavioral specification, which uses relational operators and boolean and arithmetic expressions on operation and attribute names, is not shown here.

```
Interface BankAccount {
     void deposit (in Euro amount);
     // {pre: amount > 0}
     // {post: balance() = balance()@pre + amount}
     void withdraw (in Euro amount);
     // ...
     // {invariant: balance() >= overdraftLimit()
     // {syncc: mutex (deposit, withdraw)
     ...
```

Figure 6: BankAccount interface enhanced with behavioral specifications and synchronization constraints

The `mutex` constraint serves to ensure that invocations of `deposit` and `withdraw` operations may not overlap. In [12] we have shown how such specifications of synchronization constraints can be mapped into code. This code maintains two state variables for each operation affected by such constraints. The state variables keep track of two types of events: the start of an operation execution and its termination. The code observes guards in such a way that an operation execution is deferred or rejected unless the guard is true. A guard simply relates the numbers of occurrences of these events to implement the semantics of the mutex and other synchronization operators.

4 Design of an eXtended IDL Processor

The design of a processor for eXtended IDL was inspired by previous work on a language definition environment that supported complex structure driven computations and object transformations [17]. This work relied on the old idea of syntax-directed translation schemes that were developed in the framework of parsing and compilation theory to specify mappings from one language to another. Following the structure of a language, syntax-directed translation schemes are "generation techniques ... interspersed with parsing operations" [1].

Syntax-directed translation schemes (SDTS) associate the nonterminals and production rules of two context free grammars G_1 and G_2. An SDTS defines a mapping which, given a parse tree built according to the rules of grammar G_1, determines a tree according to the rules of grammar G_2 and hence a text written in the second language. In the simplest case an implementation of a translation scheme is a pure tree manipulation. More complex implementations have interspersed parsing and unparsing operations that interpret the output terminal symbols of G_2 as calls to output actions.

The processing environment sketched in Fig. 7 is based on XPath, XSLT, and XSL. A preliminary design has been published in [14]. XPath operates on an XML document represented internally as a tree. XSL and XSLT serve to specify translation rules and output actions. XSLT provides template rules with patterns. The pattern serves to identify the XML nodes to which the template applies.

In Fig. 7 we denote the extended interface definition language by XIDL and bare IDL simply by IDL, whose XML format is defined through a DTD, denoted by `IDL.dtd` as discussed in Section 3.1. The IDL extension is captured in a separate DTD `eXtension.dtd`, which substitutes or extends selected elements in `IDL.dtd`. The merge of both DTDs constitutes the DTD of the extended interface definition language denoted by `XIDL.dtd`. The latter corresponds to G_1, while the syntax G_2 of the target programming language is implicitly defined in the XLST definition of translation rules and output actions. The XIDL processing tool transforms the input language `XIDL` to a suitable output format, such as Java stubs and skeletons, via XSLT transformation rules.

The support code implementing the IDL extensions is integrated with the stub and skeleton code according to the *extension adapter design pattern* defined in [13]. Stubs and skeletons must either be based on the dynamic invocation and

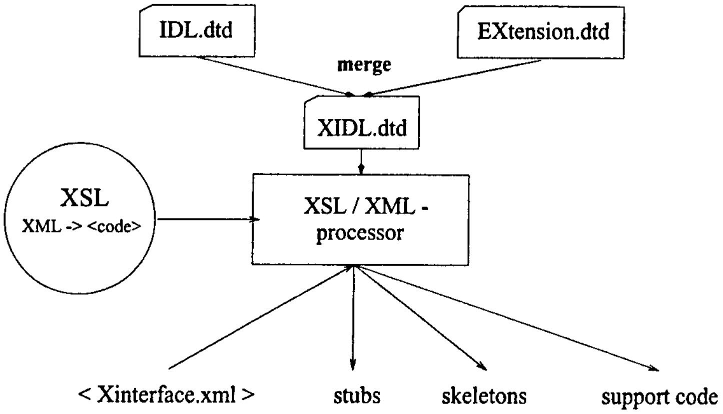


Figure 7: XML DTD processing stages to synthesize XIDL processor.

the dynamic skeleton interface or they must exploit the Java portability layer as the stub or skeleton ORB interfaces, respectively, are not specified in the CORBA standard (cf., e.g., [11] for implications and limitations).

5 Conclusion

In this paper we have introduced a framework to specify IDL and IDL extensions in a combined specification language based on XML document type definitions. We have demonstrated how this framework can be used to model different IDL extensions. Moreover, we have sketched a design of a processor for eXtensible IDL (XIDL).

We are currently working on an implementation of this framework based on standard XSLT processing tools. As outlined above, our implementation will only be able to exploit the less performant dynamic interfaces of current ORBs. Another alternative is the Java portability interface, which is only available for ORBs conformant to the standard Java language mapping. This limited range of choices is due to limitations in the openness and extensibility of the CORBA standard [11].

We intend to base our approach on the XML Schema standard[1] by replacing the IDL DTD by an XML schema definition of IDL. This will allow us to include more rigorous type information in interfaces and thus improve type checking of service interfaces expressed in XML.

Appendix: Full DTD for OMG IDL

The following constitutes a complete XML DTD for OMG IDL, compliant to CORBA 2.2 [21]. This DTD does not include the recently added language identi-

[1] See http://www.w3.org/XML/Schema

fiers, such as object-by-value argument passing, which could easily be added.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!--                                          -->
<!-- A DTD for OMG IDL, compliant to CORBA 2.2 -->
<!--                                          -->

<!ELEMENT IDL              ( COMMENT | DECLARATION | MODULE | INTERFACE )* >

<!ELEMENT COMMENT          (#PCDATA) >

<!ELEMENT DECLARATION      ( CONSTRUCTED_TYPE | CONSTANT | EXCEPTION |
                             FWD_REFRENCE )* >

<!ELEMENT CONSTRUCTED_TYPE ( STRUCTURED_TYPE | TYPEDEFED_TYPE )* >

<!ELEMENT STRUCTURED_TYPE  ( STRUCT | UNION | ENUM ) >

<!ELEMENT TYPEDEFED_TYPE   ( ARRAY | BND_SEQUENCE | UNBND_SEQUENCE | STRING |
                             TYPEDEF ) >

<!ELEMENT FWD_REFRENCE EMPTY >
<!ATTLIST FWD_REFRENCE
           name    CDATA    #REQUIRED >

<!ELEMENT EXCEPTION (MEMBER*) >
<!ATTLIST EXCEPTION
           name    CDATA    #REQUIRED >
<!ELEMENT MEMBER EMPTY >
<!ATTLIST MEMBER
           name    CDATA    #REQUIRED
           type    CDATA    #REQUIRED >

<!ELEMENT CONSTANT EMPTY >
<!ATTLIST CONSTANT
           name    CDATA    #REQUIRED
           type    CDATA    #REQUIRED
           value   CDATA    #REQUIRED >

<!ELEMENT STRUCT (MEMBER*) >
<!ATTLIST STRUCT
           name     CDATA    #REQUIRED >

<!ELEMENT UNION (MEMBER*) >
<!ATTLIST UNION
           name     CDATA    #REQUIRED
           sw_type CDATA    #REQUIRED >

<!ELEMENT ENUM (ELEMENT+) >
<!ATTLIST ENUM
           name     CDATA    #REQUIRED >
<!ELEMENT ELEMENT EMPTY >
<!ATTLIST ELEMENT
           value     CDATA    #REQUIRED >

<!ELEMENT ARRAY (DIMENSION+) >
<!ATTLIST ARRAY
           name    CDATA    #REQUIRED
           type    CDATA    #REQUIRED >
<!ELEMENT DIMENSION EMPTY >
<!ATTLIST ARRAY
           value   CDATA    #REQUIRED >
```

```
<!ELEMENT BND_SEQUENCE EMPTY >
<!ATTLIST BND_SEQUENCE
           name  CDATA   #REQUIRED
           type  CDATA   #REQUIRED
           bound CDATA   #REQUIRED >

<!ELEMENT UNBND_SEQUENCE EMPTY >
<!ATTLIST UNBND_SEQUENCE
           name  CDATA   #REQUIRED
           type  CDATA   #REQUIRED >

<!ELEMENT STRING EMPTY>
<!ATTLIST STRING
           name    CDATA   #REQUIRED
           lenght  CDATA   #REQUIRED >

<!ELEMENT TYPEDEF EMPTY >
<!ATTLIST TYPEDEF
           typename  CDATA   #REQUIRED
           name      CDATA   #REQUIRED >

<!ELEMENT MODULE  ( DECLARATION | INTERFACE )* >
<!ATTLIST MODULE
           name    CDATA   #REQUIRED >

<!ELEMENT INTERFACE  ( INHERITANCE*, ( DECLARATION | ATTRIBUTE | SIGNATURE )* ) >
<!ATTLIST INTERFACE
           name    CDATA   #REQUIRED >

<!ELEMENT INHERITANCE EMPTY >
<!ATTLIST INHERITANCE
           ancestor   CDATA   #REQUIRED >

<!ELEMENT ATTRIBUTE EMPTY >
<!ATTLIST ATTRIBUTE
          mode    ( readonly  |  readwrite )  "readwrite"
          name    CDATA   #REQUIRED
          type    CDATA   #REQUIRED  >

<!ELEMENT SIGNATURE  ( ARGUMENT*, ( RAISES? ), ( CONTEXT? ) ) >
<!ATTLIST SIGNATURE
          mode    ( oneway  |  twoway )  "twoway"
          name    CDATA   #REQUIRED
          rtype   CDATA   #REQUIRED  >

<!ELEMENT ARGUMENT   EMPTY >
<!ATTLIST ARGUMENT
          mode     ( in | out | inout ) #REQUIRED
          name     CDATA                #REQUIRED
          type     CDATA                #REQUIRED >

<!ELEMENT RAISES (EXCEPTION_TYPE+) >
<!ELEMENT EXCEPTION_TYPE  EMPTY >
<!ATTLIST EXCEPTION_TYPE
           exception     CDATA          #REQUIRED >

<!ELEMENT CONTEXT  (CONTEXT_ELEMENT+) >
<!ELEMENT CONTEXT_ELEMENT EMPTY >
<!ATTLIST CONTEXT_ELEMENT
           name     CDATA                #REQUIRED >
```

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall, 1972

[2] C. Becker and K. Geihs. Generic QoS specifications for CORBA. In *Kommunikation in Verteilten Systemen (KIVS'99)*, pages 184–195. Springer-Verlag, 1999.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (editors). Extensible markup language (XML) 1.0. Technical report, W3C, Feb. 1998.

[5] B. Bukowski. Interaction protocols: Typing of object interactions in frameworks. Technical report TR B 96–10, Freie Universität, Berlin, 1996.

[6] C. Della, T. Cicalese, and S. Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, July 1999.

[7] S. Frølund. *Coordinating Distributed Objects*. MIT Press, 1996.

[8] H. Gruender and K. Geihs. On the object-oriented modelling of distributed workflow applications. In *3. Internationale Tagung Wirtschaftsinformatik (WI)*, Berlin, Germany, 1997.

[9] D. Hagimont, J. Mossire, X. Rousset de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 282–289, May 1996.

[10] O. Holder, L. Ben-Schaul, and H. Gazir. Dynamic layout of distributed applications in Fargo. Technical report, Technion – Israel Institute of Technology, Aug 1998.

[11] H.-A. Jacobsen. Programming language interoperability in distributed computing environments. In Lea Kutvonen, Harmunt Knig, and Martti Tienari, editors, *Second IFIP Working Conference on Distributed Applications and Interoperable Systems II (DAIS)*, Helsinki, Finland, June 1999. Kluwer Academic Publisher.

[12] H.-A. Jacobsen and B. J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *IEEE Automated Software Engineering Conference (ASE'98)*, pages 63–72. IEEE Computer Society, September 1998.

[13] H.-A. Jacobsen and B. Krämer. Design patterns for synchronization adaptors of CORBA objects. *Special issue of L'OBJET: Journal on "Object Orientation and Formal Methods*, Hermes Publisher, 2000.

[14] H.-A. Jacobsen and B. J. Krämer. Modeling Interface Definition Language Extensions. In *Technology of Object-Oriented Languages and Systems (TOOLS-Pacific 2000)*, pages 242–252. IEEE Computer Society, November 2000.

[15] K. Keahey and D. Gannon. PARDIS: A parallel approach to CORBA. In *International Symposium on High Performance Distributed Computing*, pages 31–39. IEEE Computer Society, Aug 1997.

[16] Bernd J. Krämer. *Synchronization Constraints in Object Interfaces*, volume Information Systems Interoperability, chapter 5. Research Studies Press, 1998.

[17] Bernd J. Krämer and H.-W. Schmidt. *Developing Integrated Environments with ASDL, IEEE Software*, January 1989, pp. 98–106, IEEE Computer Society.

[18] Jeff Magee and Jeff Kramer. *Composing Distributed Objects in CORBA*, volume Information Systems Interoperability, chapter 7. Research Studies Press, 1998.

[19] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[20] OMG. CORBA components RFP. Technical report, Object Management Group, 1998. URL: http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html.

[21] OMG. The Common Object Request Broker Architecture and Specification. Revision 2.0. Technical report, Object Management Group, 1998.

[22] A. Patzke. Der Weg von IDL zu XML: IDL2XML-Compiler. Technical report, University of Technology Hamburg-Harburg Technische Informatik 5 (FSP 4-10) Schwarzenbergstr. 95, July 1998. (in German) URL: http://kant.ti5.tu-harburg.de/Publication/1998/reports/idl2xml/-default.htm.

[23] A. Puder. A declarative extension of IDL-based type definitions within open distributed environments. In *International Conference on Object Oriented Information Systems*, pages 423–436, 1998.

[24] S. Sankar and R. Hayes. An Interface Definition Language for Specifying and Testing Software. *ACM Sigplan Notices*, 29(8), Aug 1994.

[25] D. C. Schmidt, D. Levine, and S. Mungee. The design of the Tao real-time object request broker. *Computer Communications*, 21(4), 1998.

[26] von de las Heras Quiros and Olmo Millan. Inheritance Anomaly in CORBA Multithreaded Environments. *Theory and Practice of Object Systems*, 3(1):45–54; 1997.

[27] D. Watkins. Using interface definition languages to support path expressions and programming by contract. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, 1998.

[28] Andrew Watson. CORBA and XML; conflict or cooperation? Technical report, Object Management Group, 1999. URL: http://www.omg.org/library/watsonwp.html.

[29] V. F. Wolfe, J. Black, B. Thuraisingham, and P. Krupp. Real-time method invocations in distributed environments. Technical report, University of Rhode Island, Jan 1996.

[30] X/Open SUN Microsystems. *ADL Language Reference Manual (1.0 edition)*, 1996. URL: http://www.sunlabs.com/research/adl.

[31] V. Zadorozhny. Towards an integrated CORBA/RAISE semantic interoperable environment. Technical Report 117, UNU/IIST, P.O.Box 3058, Macau, July 1997.

[32] J. A. Zinkey, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):56–73, 1997.

Subclassing errors, OOP, and practically checkable rules to prevent them

Oleg Kiselyov
Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract

This paper considers an example of Object-Oriented Programming (OOP) leading to subtle errors that break separation of interface and implementations. A comprehensive principle that guards against such errors is undecidable. The paper introduces a set of *mechanically verifiable* rules that prevent these insidious problems. Although the rules seem restrictive, they are powerful and expressive, as we show on several familiar examples. The rules contradict both the spirit and the letter of the OOP. The present examples as well as available theoretical and experimental results pose a question if OOP is conducive to software development at all.

Keywords: object-oriented programming, subtyping, subclassing, implementation inheritance, C++, functional programming

1 Introduction

Decoupling of abstraction from implementation is one of the holy grails of good design. Object-oriented programming is claimed to be conducive to such a separation, and therefore to more reliable code. In the end, productivity and quality are the only true merits a programming methodology is to be judged upon. This article will discuss a simple example that questions if Object-Oriented Programming (OOP) indeed helps separate interface from implementation. First we demonstrate how easily subclassing errors arise and how difficult (in general, undecidable) it is to prevent them. We later introduce a set of expressive rules that preclude the subclassing errors, and can be mechanically verified. Incidentally the rules run contrary to the OOP precepts.

We take a rather familiar example that illustrates the difference between subclassing and subtyping: the example of Sets and Bags. The example is isomorphic to that of circles vs. ellipses or squares vs. rectangles. Section 2 introduces the example and carries it one step further, to a rather unsettling result: a "transparent" change in an implementation suddenly breaks client code that was written according to public interfaces. We set out to follow good software engineering practices; this makes the resulting failure even more ominous. Section 3 brings up a subclassing vs. subtyping dichotomy and the Liskov principle of behavioral substitutability. We show that Sets and Bags viewed as mutable or immutable *objects* are not subtypes of each other. The indiscriminate use of implementation inheritance indeed prevents separation of interface and implementation. In Section 4 we take a contrary point of view, of bags and sets as values without a hidden state and whose responses to external messages cannot be overridden. We prove that a set truly *is-a* bag; a set *is* substitutable for a bag, a set can always be manipulated as a bag, a set maintains every invariant of a bag – and it is still a set. The section also shows that if we abide by practically checkable rules we obtain a guarantee that the subtle subclassing errors cannot occur in principle. We will also show that the rules do not diminish the power of a language.

Inheritance and encapsulation, two staples of OOP, make checking of the Liskov Substitution Principle (LSP) for derived objects generally undecidable. On the other hand, the proposed rules, which *can* be checked at compile time, make derived values satisfy LSP.

The article aims to give a more-or-less "real" example, which we can run and see the result for ourselves. By necessity the example had to be implemented in some language. The present article uses C++. It appears however that similar code and similar conclusions can be carried on in many other object-oriented languages (e.g., Java, Python, etc).

2 Coupling of interface and implementation

Suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). I chose the following interface:

```
typedef int const * CollIterator;      // Primitive but will do
class CBag {
 public:
  int size(void) const;
  int count(const int elem) const;
  virtual void put(const int elem);
  virtual bool del(const int elem);
  CollIterator begin(void) const;
  CollIterator end(void) const;

  CBag(void);
  virtual CBag * clone(void) const;
 private: ...        // implementation details elided
};
```

The class CBag defines usual methods to determine the number of all elements in a bag, to count the number of occurrences of a specific element, to put a new element into a bag and to remove one. The latter function returns false if the element to delete did not exist. We also define the standard enumerator interface [11] – methods begin() and end() – and a method to make a copy of the bag. Other operations of the CBag package are implemented without the knowledge of CBag's internals: the print-on operator <<, the union (merge) operator +=, and operators to compare CBags and to determine their structural equivalence. These functions use only the public interface of the CBag class:

```
void operator += (CBag& to, const CBag& from);
bool operator <= (const CBag& a, const CBag& b);
inline bool operator >= (const CBag& a, const CBag& b)
{ return b <= a; }
inline bool operator == (const CBag& a, const CBag& b)
{ return a <= b && a >= b; }
```

The complete code of the whole example is available in [7]. It has to be stressed that the package was designed to minimize the number of functions that need to know details of CBag's implementation. Following good practice, I wrote validation code (file vCBag.cc [7]) that tests all the functions and methods of the CBag package and verifies common invariants.

Suppose you are tasked with implementing a Set package. Your boss defined a set as an unordered collection where each element has a single occurrence. In fact, your boss even said that a set is a bag with no duplicates. You have found my CBag package and realized that it can be used with few additional changes. The definition of a Set as a Bag, with some constraints, made the decision to reuse the CBag code even easier.

```
class CSet : public CBag {
 public:
  bool memberof(const int elem) const
  { return count(elem) > 0; }

  // Overriding of CBag::put
  void put(const int elem)
  { if(!memberof(elem)) CBag::put(elem); }

  CSet * clone(void) const
  { CSet * new_set = new CSet();
    *new_set += *this; return new_set; }
  CSet(void) {}
};
```

The definition of a CSet makes it possible to mix CSets and CBags, as in set += bag; or bag += set; These operations are well-defined, keeping in mind that a set is a bag that happens to have the count of all members exactly one. For example, set += bag; adds all elements from a bag to a set, unless they are already present. On the other hand, bag += set; is no different than merging a bag with any other bag. You too wrote a validation suite to test all CSet methods (newly defined as well as inherited from a bag) and to verify common expected properties, e.g., a+=a ≡ a.

In my package, I have defined and implemented a function that, given three bags a, b, and c, decides if a+b is a subbag of c:

```
bool foo(const CBag& a, const CBag& b, const CBag& c)
{                     // Clone a to avoid clobbering it
  CBag & ab = *(a.clone());
  ab += b;            // ab is now the union of a and b
  bool result = ab <= c;
  delete &ab;
  return result;
}
```

It was verified in the test suite. You have tried this function on sets, and found it satisfactory.

Later on, I revisited my code and found my implementation of foo() inefficient. Memory for the ab object is unnecessarily allocated on heap. I rewrote the function as

```
bool foo(const CBag& a, const CBag& b, const CBag& c)
{
  CBag ab;
  ab += a;           // Clone a to avoid clobbering it
  ab += b;           // ab is now the union of a and b
  bool result = ab <= c;
  return result;
}
```

It has exactly the same interface as the original foo(). The code hardly changed. The behavior of the new implementation is also the same – as far as I and the package CBag are concerned. Remember, I have no idea that you are re-using my package. I re-ran the validation test suite with the new foo(): everything tested fine.

However, when you run your code with the new implementation of foo(), you notice that something *has* changed! The complete source code [7] contains tests that make this point obvious: Commands make vCBag1 and make vCBag2 run validation tests with the first and the second implementations of foo(). Both tests complete successfully, with the identical results. Commands make vCSet1 and make vCSet2 test the CSet package. The tests – other than those of foo() – all succeed. Function foo() however yields markedly different results. It is debatable which implementation of foo() gives truer results for CSets. In any case, changing internal algorithms of a pure function foo() while keeping the same interfaces is not supposed to break your code. What happened?

What makes this problem more unsettling is that both you and I tried to do everything by the book. We wrote a safe, typechecked code. We eschewed casts. g++ (2.95.2) compiler with flags -W and -Wall issued not a single warning. Normally these flags cause g++ to become very annoying. You did not try to override methods of CBag to deliberately break the CBag package. You attempted to preserve CBag's invariants (weakening a few as needed). Real-life classes usually have far more obscure algebraic properties. We both wrote validation tests for our implementations of a CBag and a CSet, and they passed. And yet, despite all my efforts to separate interface and implementation, I failed. Should a programming language or the methodology take at least a part of the blame? [10, 4, 1]

3 Subtyping vs. Subclassing

The breach of separation between CBag's implementation and interface is caused by CSet design's violating the Liskov Substitution Principle (LSP) [9]. CSet has been declared a subclass of CBag. Therefore, C++ compiler's typechecker permits passing a CSet object or a CSet reference to a function that expects a CBag object or reference. However, it is well known [3] that a CSet is not a *subtype* of a CBag. The next few paragraphs give a simple proof of this fact, for the sake of reference.

The previous section considered bags and sets from the OOP perspective – as objects that encapsulate state and behavior. Behavior means an object can accept a message, send a reply and possibly change its state. From this point of view, bags and sets are not subtypes of each other. Indeed, let us define a Bag as an object that accepts two messages: (send a-Bag 'put x) puts an element x into the Bag, and (send a-Bag 'count x) gives the occurrence count for x in the Bag (without changing a-Bag's state). Likewise, a Set is defined as an object that accepts two messages: (send a-Set 'put x) puts an element x into a-Set unless it was already there, (send a-Set 'count x) gives the count of occurrences of x in a-Set (which is always either 0 or 1). Throughout this section we use a different, concise notation to emphasize the general nature of the argument.

Let us consider a function

```
(define (fnb bag) (send bag 'put 5) (send bag 'put 5) (send bag 'count 5))
```

The behavior of this function, its contract, can be summed as follows: given a Bag, the function adds two elements into it and returns `(+ 2 (send orig-bag 'count 5))`. Technically you can pass to `fnb` a Set object as well. Just as a Bag, a Set object accepts messages `'put` and `'count`. However applying `fnb` to a Set object will break the function's post-condition stated above. Therefore, passing a set object where a bag was expected changes the behavior of a program. According to the LSP, a Set is not substitutable for a Bag – a Set cannot be a subtype of a Bag.

Let us consider another function

```
(define (fns set) (send set 'put 5) (send set 'count 5))
```

The behavior of this function is: given a Set, the function adds an element into it and returns 1. If you pass to this function a bag (which – just as a set – replies to messages `'put` and `'count`), the function `fns` may return a number greater than 1. This will break `fns`'s contract, which promised always to return 1.

One may claim that "A Set is not a Bag, but an ImmutableSet is an ImmutableBag." This is not correct either. An immutability per se does not confer subtyping to "derived" classes of data, as a variation of the previous argument shows [8]. C++ objects are record-based. Subclassing is a way of extending records, with possibly altering some slots in the parent record. Those slots must be designated as modifiable by a keyword `virtual`. In this context, prohibiting mutation and overriding makes subclassing imply subtyping. This is the reasoning behind BRules introduced below. However merely declaring the state of an object immutable is not enough to guarantee that derivation leads to subtyping: An object can override parent's behavior without altering the parent. This is easy to do when an object is implemented as a functional closure, when a handler for an incoming message is located with the help of some kind of reflexive facilities, or in prototype-based OO systems [8]. Incidently, if we do permit a derived object to alter its base object, we implicitly allow behavior overriding. For example, an object `A` can react to a message `M` by forwarding the message to an object `B` stored in `A`'s slot. If an object `C` derived from `A` alters that slot it hence overrides `A`'s behavior with respect to `M`.

The OOP point of view thus leads to a conclusion that neither a Bag nor a Set are subtypes of the other. The interface or an implementation of a Bag and a Set appear to invite subclassing of a Set from a Bag, or vice versa. Doing so however will violate the LSP – and we have to brace for strikingly subtle errors. The previous section intentionally broke the LSP to demonstrate how insidious the errors are and how difficult it may be to find them. Sets and Bags are very simple types, far simpler than the ones that typically appear in a production code. Since LSP when considered from an OOP point of view is undecidable, we cannot count on a compiler for help in pointing out an error. As Section 2 showed, we cannot rely on validation tests either. We have to *see* the problem [4, 10, 1].

4 Mechanically preventing subclassing errors

Bags and sets – as objects – indeed are not subtypes. Subclassing them violates LSP, which leads to insidious errors. Bags and sets however do not have to be viewed as objects. We can take them as pure values, without any state or intrinsic behavior – just like the numbers are. In Section 2, CBag and CSet objects encapsulated a hidden state – a collection of integers. The objects had an ability to react to messages, e.g., `put` and `del`, by altering their state. In this section we re-do the example of Section 2 using a different approach. Bags and sets no longer have a state that is distinct from their identity and that can be altered. Equally important we do not allow any changes to the behavior of bags and sets with respect to applicable operations, by overriding or otherwise. In other words, every post-condition of a bag or a set constructor holds throughout the lifespan of the constructed values. This approach makes the subclassing problems and breach of encapsulation disappear. It turns out that a set truly *is-a* bag; a set is substitutable for a bag, a set can always be manipulated as a bag, a set maintains every invariant of a bag – and it is still a set.

The LSP says, "If for each object o1 of type S there is another object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T." If type T denotes a set of values that carry their own behavior, and if values of type S can override some of T values behavior, the LSP is undecidable. Indeed, a mechanical application of LSP must at least be able to verify that all methods overridden in S terminate whenever the corresponding methods in T terminate. This is generally impossible. On the other hand, if T denotes a set of (structured) data values, and S is a subset of these values – e.g., restricted by range, parity, etc. – the LSP is trivially satisfied.

This section also shows that if one abides by mechanically verifiable rules he obtains a guarantee that the subtle subclassing errors cannot occur in principle. The rules do not reduce the power of a language.

4.1 BRules

Suppose I was given a task to implement a Bag – an unordered collection of possibly duplicate items (integers in this example). This time my boss laid out the rules, which we will refer to as *BRules*:

- no virtual methods or virtual inheritance
- no visible members or methods in any public data structure (that is, in any class declared in an .h file)
- no mutations to public data structures
  - a strict form: no assignments or mutations whatsoever
  - a less strict form: no function may alter, directly or indirectly, any data it receives as arguments

The rules break the major tenets of OOP: for example, values no longer have a state that is separate from their identity. Prohibitions on virtual methods and on modifications of public objects are severe. It appears that not much of C++ is left. Surprisingly I still can implement my assignment without losing expressiveness – and perhaps even gaining some. The exercise will also illustrate that C++ does indeed have a pure functional subset [12], and that you can program in C++ without assignments.

4.2 Interface and implementation of a FBag

```
class FBag {
 public:
  FBag(void);
  FBag(const FBag& another);    // Copy-constructor
  ~FBag(void);

 private:
  class Cell;                   // Opaque type
  const Cell * const head;
  FBag(const Cell * const cell); // Private constructor
                // Declaration of three friends elided
};
```

Indeed, there are no virtual functions, no methods or public members. We also declare functions that take a FBag as (one of the) arguments and return the count of all elements or a specific element in the bag, print the bag, *fold* [5] over the bag, compare two bags for structural equivalence, verify bag's invariants, merge two bags, add or delete an element. The latter three functions do not modify their arguments; they return a new FBag as their result. It must be stressed that the functions that operate on a FBag are not FBag's methods; in particular, they are not a part of the class FBag, they are not inherited and they cannot be overridden. The implementation is also written in a functional style. FBag's elements are held in a linked list of cells, which are allocated from a pre-defined pool. The pool implements a mark-and-sweep garbage collection, in C++.

Forgoing assignments does not reduce expressiveness as the following snippet from the FBag code shows; the snippet implements the union of two FBags:

```
struct union_f {
  FBag operator() (const int elem, const FBag seed) const {
    return put(seed,elem);
  }
};
FBag operator + (const FBag& bag1, const FBag& bag2)
{
  return fold(bag1,union_f(),bag2);
}
```

Following good practice, I wrote a validation code (file vFBag.cc [7]) that tests all the functions of the FBag package and verifies common invariants.

4.3 Implementation of a FSet. FSet is a subtype of a FBag

Suppose you are tasked with implementing a Set package. Your boss defined a set as an unordered collection where each element has a single occurrence. In fact, your boss even said that a set is a bag with no duplicates. You have found my FBag package and realized that it can be used with few additional changes. The definition of a Set as a Bag (with some constraints) made the decision to reuse the FBag code even easier.

```
class FSet : public FBag {
 public:
  FSet(void) {}
  FSet(const FBag& bag) : FBag(remove_duplicates(bag)) {}
};

bool memberof(const FSet& set, const int elem)
{ return count(set,elem) > 0; }
```

Surprisingly, this is the *whole* implementation of a FSet. A set is fully a bag. Because FSet constructors eventually call FBag constructors and do no alter the latter's result, every post-condition of a FSet constructor implies a post-condition of a FBag constructor. Since FBag and FSet values are immutable, the post-conditions that hold at their birth remain true through their lifespan. Because all FSet values are created by an FBag constructor, all FBag operations automatically apply to an FSet value. This concludes the proof that an FSet is a *subtype* of a FBag.

The `FBag.cc` package [7] has a function `verify(const FBag&)` that checks to make sure its argument is indeed a bag. The function tests FBag's invariants, for example:

```
const FBag bagnew = put(put(bag,5),5);
assert( count(bagnew,5) == 2 + count(bag,5) &&
        size(bagnew) == 2 + size(bag) );
assert( count(del(bagnew,5),5) == 1 + count(bag,5) );
```

Your validation code passes a non-empty set to this function to verify the set is indeed a bag. You can run the validation code `vFSet.cc` [7] to see for yourself that the test passes. On the other hand, FSets do behave like Sets:

```
const FSet a112 = put(put(put(FSet(),1),1),2);
assert( count(a112,1) == 1 );

const FSet donce = FSet() + a112;
const FSet dtwice = donce + a112;
assert( dtwice == a112 );
```

where `a112` is a non-empty set. The validation code `vFSet.cc` you wrote contains many more tests like the above. The code shows that a FSet is able to pass all of FBag's tests as well as its own. The implementation of FSets makes it possible to take a union of a set and a bag; the result is always a bag, which can be made a set if desired. There are corresponding test cases as well.

To clarify how an FSet may be an FBag at the same time, let us consider one example in more detail:

```
    // An illustration that an FSet is an FBag
int cntb(const FBag v) {
  FBag b1 = put(v, 5);  FBag b2 = put(b1, 5);
  FBag b3 = del(b2, 5);
  return count(b3, 5); }
const int cb1 = cntb(FBag()); // cb1 has the value of 1
const int cb2 = cntb(FSet()); // cb2 has the value of 1

    // An illustration that an FSet does act as a set
int cnts(const FSet v) {
  FSet s1 = put(v, 5); FSet s2 = put(s1, 5);
  FSet s3 = del(s2, 5);
  return count(s3, 5); }
const int cs = cnts(FSet()); // cs has the value of 0
```

This example is one of the test cases in `vFSet.cc` [7]. You can run it and check the results for yourself. Yet it is puzzling: how come `cs` has the value different from that of `cb1` if there is no custom `del()` function for FSets? The statement `FSet s2 = put(s1, 5);` is the most illuminating. On the right-hand side is an expression: putting an element 5 to a FBag/FSet that already has this element in it. The result of that expression is a FBag {5,5}, with two instances of element 5. The statement then constructs a FSet `s2` from that bag. A FSet constructor is invoked. The constructor takes the bag {5,5}, removes the duplicate element 5 from it, and "blesses" the resulting FBag to be a FSet as well. Thus `s2` will be a FBag and a FSet, with one instance of element 5. In fact, `s1` and `s2` are identical. A FSet constructor guarantees that a FBag it constructs contains no duplicates. As objects are immutable, this invariant holds forever.

4.4 Discussion

Surprising as it may be, assertions "a Set is a Bag with no duplicates" and "a Set always acts as a Bag" do not contradict each other, as the following two examples illustrate:

Let `{value ...}` be an unordered collection of values: a Bag. Let us consider the following values: $vA : 42$, $vB : \{42\}$, $vC : \{43\}$, $vD : \{42\,43\}$, $vE : \{42\,43\,42\}$ vA is not a collection; vB, vC, vD, and vE are bags. vB, vC, and vD are also Sets: unordered collections without duplicates. vE is not a Set. Every Set is a Bag but not every Bag is a Set.	Let *uf-integer* denote a natural number whose prime factors are unique. Let us consider the following values: $vA : \frac{5}{4}$, $vB : 42$, $vC : 43$, $vD : 1806$, $vE : 75852$ vA is not an integer; vB, vC, vD, and vE are integers. vB, vC, and vD are also uf-integers. vE is not a uf-integer as it is a product $2 * 2 * 3 * 3 * 7 * 7 * 43$ with factors 2, 3, and 7 occurring several times. Every uf-integer is an integer but not every integer is a uf-integer.
We introduce operations *merge* (infix +) and *subtract* (infix -). Both operations take two Bags and return a Bag. Either of the operands, or both, may also be a Set. The result, a Bag, may or may not be a Set. For example, $vB + vC \Rightarrow vD$ Both of the operands and the result are also Sets $vB + vD \Rightarrow vE$ The argument Bags are also Sets, but the resulting Bag is not a Set $vE + vE \Rightarrow \{42\,43\,42\,42\,43\,42\}$ None of the Bags here are Sets $vD - vC \Rightarrow vB$ The argument Bags are also Sets, so is the result. $vE - vC \Rightarrow \{42\,42\}$ One of the arguments is a Set, the resulting Bag is not a Set. $vE - vE \Rightarrow \{\}$ The argument Bags are not Sets, but the resulting Bag is.	We introduce operations *multiply* (infix *) and *reduce* (infix %): $a\%b = a/gcd(a, b)$. Both operations take two integers and return an integer. Either of the operands, or both, may also be a uf-integer. The result, an integer, may or may not be a uf-integer. For example, $vB * vC \Rightarrow vD$ Both of the operands and the result are also uf-integers $vB * vD \Rightarrow vE$ The argument integers are also uf-integers, but the resulting integer is not a uf-integer $vE * vE \Rightarrow 5753525904$ None of the integers here are uf-integers $vD\%vC \Rightarrow vB$ The argument integers are also uf-integers, so is the result $vE\%vC \Rightarrow 1764$ One of the arguments is a uf-integer, the resulting integer is not a uf-integer $vE\%vE \Rightarrow 1$ The argument integers are not uf-integers, but the resulting integer is.

Bags are closed under operation *merge* but subsets of Bags – Sets – are not *not* closed under *merge*. On the other hand, both Bags and Sets are closed under *subtract*. We may wish for a merge-like operation that, being applied to Sets, always yields a Set. We can introduce a new operation: $merge - if - not - there$. We can define it specifically for Sets. Alternatively, the operation can be defined on Bags; it would apply to Sets by the virtue of inclusion polymorphism as every Set is a Bag. Sets *are* closed with respect to $merge-if-not-there$. On the other hand, to achieve closure of Sets under *merge* we can project – coerce – the result of merging of two Sets back into Sets, a subset of Bags. The FBag/FSet package took this approach. If we *merge* two FSets and want to get an FSet in result we have to specifically say so, by applying a projection (coercion) operator: `FSet::FSet(const FBag& bag)`. That operator creates a new FBag without duplicates. This fact makes the latter a FSet. Thus $FSet(vB + vD) \Rightarrow vD$, an FSet.	Integers are closed under operation *multiply* but subsets of integers – uf-integers – are *not* closed under *multiply*. On the other hand, both integers and uf-integers are closed under *reduce*. We may wish for a multiply-like operation that, being applied to uf-integers, always yields a uf-integer. We can introduce a new operation: *lcm*, the least common multiple. This operation is well-defined on integers; it would apply to uf-integers by the virtue of inclusion polymorphism as every uf-integer is an integer. uf-integers *are* closed with respect to the *lcm* operation. On the other hand, to achieve closure of uf-integers under *multiply* we can project – coerce – the product of two uf-integers back into uf-integers, a subset of integers. If we *multiply* two uf-integers and want to get a uf-integer in result we have to specifically say so, by applying a projection (coercion) operator: $remove - duplicate - factors$. That operator creates a new integer without duplicate factors. This fact makes the resulting integer a uf-integer. Thus $uf-integer(vB*vD) \Rightarrow vD$, a uf-integer

It has to be stressed that the two columns of the above table are not merely similar: they are isomorphic. Indeed, the right column is derived from the left column by the following substitution of words that preserves meaning: Bag ↔ integer, Set ↔ uf-integer, merge ↔ multiply, subtract ↔ reduce. The right column sounds more "natural" – so should the left column as integers and uf-integers are representations for resp. FBags and FSets.

From an extensional point of view [2], a type denotes a set of values. By definition of a FSet, it is a particular kind of FBag. Therefore, a set of all FSets is a subset of all FBags: FSet is a subtype of FBag. A FBag or a FSet do not have any "embedded" behavior – just as integers do not have an embedded behavior. Behavior of numbers is defined by operations, mapping from numbers to numbers. Any function that claims to accept every member of a set of values identified by a type T will also accept any value in a subset of T. Frequently a value can participate in several sets of operations: a value can have several types at the same time. For example, a collection { 42 } is both a Bag and a Set. This fact should not be surprising. In C++, a value typically denoted by a numeral 0 can be considered to have a character type, an integer type, a float type, a complex number type, or a pointer type, for any declared or yet to be declared pointer type. This lack of behavior is what puts FBag and FSet apart from CBag and CSet discussed in the previous article. FSet is indeed a subtype of FBag, while CSet is not a subtype of a CBag as CSet has a different behavior. Incidentally LSP is trivially satisfied for values that do not carry their own behavior. FBags and FSets are close to so-called predicate classes. Since instances of FSets are immutable, the predicate needs to be checked only at a value construction time.

4.5 Polymorphic programming with BRules

The FSet/FBag example above showed BRules in the context of subtypes formed by a restriction on a base type. As it turns out, BRules work equally well with existential (abstract) types. To illustrate this point, the source code accompanying this article [7] contains three implementations of a collection of polymorphic values. The collection is populated by Rectangles and Ellipses, which are instances of concrete classes implementing a common abstract base class Shape. A Shape is an existential type that knows how to draw, move and resize itself. A file Shapes-oop.cc gives the conventional, OOP-like implementation, with virtual functions and such. A file Shapes-no-oop.cc is another implementation, also in C++. The latter follows BRules, has no assignments or virtual functions. Any particular Shape value is created by a Shape constructor and is not altered after that. Shapes-no-oop.cc achieves polymorphic programming with the full separation of interface and implementation: If an implementation of a concrete Shape is changed, the code that constructs and uses Shapes does not even have to be recompiled! The file defines two concrete instances of the Shape: a Square and a Rectangle. The absence of mutations and virtual functions guarantees that any post-condition of a Square or a Rectangle constructor implies the post-condition of a Shape. Both particular shapes can be passed to a function `set_dim(const Shape& shape, const float width, const float height);` Depending on the new dimensions, a square can *become* a rectangle or a rectangle square. You can compile Shapes-no-oop.cc and run it to see that fact for yourself.

It is instructive to compare Shapes-no-oop.cc with Shapes-h.hs, which implements the same problem in a purely functional, strongly-typed language Haskell. All three code files in the Shapes directory solve the same problem the same way. Two C++ code files – Shapes-oop.cc and Shapes-no-oop.cc – look rather different. On the other hand, the purely functional Shapes-no-oop.cc and the Haskell code Shapes-h.hs are uncanny similar – in some places, frighteningly similar. This exercise shows that BRules do not constrain the power of a language even when abstract data types are involved.

5 Conclusions

It is known, albeit not so well, that following the OOP letter and practice may lead to insidious errors [10, 1]. Section 2 of this article showed how subtle the errors can be even in simple cases. In theory, there are rules – LSP – that could prevent the errors. Alas, the rules are in general undecidable and not *practically reinforceable.*

In contrast, BRules introduced in this article can be statically checked at compile time. The rules outlaw certain syntactic constructions (for example, assignments in some contexts, and non-private methods) and keywords (e.g., `virtual`). It is quite straightforward to write a lint-like application that scans source code files and reports if they conform to the rules. When BRules are in effect, subtle subclassing errors like the ones shown in Section 2 become impossible. To be more precise, with BRules, *subclassing implies subtyping.* Subclassing by definition is a way of creating (derived) values by extending, restricting, or otherwise specializing other, parent values. A derived value constructor must invoke a parent value constructor to produce the parent value. The former constructor often has a chance to alter the parent constructor's result before it is cast or incorporated into the derived value. If this chance is taken away, the post-condition of a derived value constructor implies the post-condition of the parent value. Disallowing any further mutations guarantees the behavioral substitutability of derived values for parent values at all times.

As the examples in this article showed, following BRules does not diminish the power of the language. We can still benefit from polymorphism, we can still develop practically relevant code. Yet BRules blur the distinction between the identity and the state, a characteristic of objects. BRules are at odds with the practice if not the very mentality of OOP. This begs the question: Is OOP indeed conducive to software development?

One can argue that OOP – as every powerful technique – requires extreme care: knives are sharp. Likewise, goto is expressive, and assembler- or microcode-level programming are very efficient. All of them can lead to bugs that are very difficult, statically impossible, to find. On the other hand, if you program, for example, in Scheme, you never have to deal with an "invalid opcode" exception. That error becomes simply impossible. Furthermore, "while opinions concerning the benefits of OOSD [Object-Oriented Software Development] abound in OO literature, there is little empirical proof of its superiority" [6].

Acknowledgments

I am grateful to Valdis Berzins for valuable discussions and suggestions on improving the presentation. This work has been supported in part by the National Research Council Research Associateship Program, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

References

[1] Cardelli, L. Bad Engineering Properties of Object-Oriented Languages. ACM Comp. Surveys 28(4es), 1996, article 150.

[2] Cardelli, L., Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. ACM Comp. Surveys, 17(4): December 1985, pp. 471-522.

[3] Cook, W.R., Hill, W.L., Canning, P.S. Inheritance Is Not Subtyping. In: Carl A. Gunter and John C. Mitchell. Theoretical Aspects of Object- Oriented Programming. MIT Press. ISBN 0-262-07155-X.

[4] Hatton, L. Does OO sync with how we think? IEEE Software 15(3) , May-June 1998, pp. 46 -54.

[5] Hutton, G. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4):355-372, July 1999.

[6] Johnson, R.A. The Ups and Downs of Object-Oriented Systems Development. Comm. ACM 43(10), October 2000, pp. 69-73.

[7] Kiselyov, O. Complete code that accompanies the article. <http://pobox.com/~oleg/ftp/packages/subclassing–problem.tar.gz>, August 4, 2000.

[8] Kiselyov, O. Subtyping, Subclassing, and Trouble with OOP. <http://pobox.com/~oleg/ftp/Computation/Subtyping/index.html>, August 4, 2000.

[9] Liskov, B., Wing, J. M. A Behavioral Notion of Subtyping. ACM Trans. Programming Languages and Systems, 16(6), November 1994, pp. 1811-1841.

[10] Ousterhout, J.K. Scripting: Higher-Level Programming for the 21st Century. IEEE Computer, March 1998, pp. 23-30.

[11] Standard Template Library Programmer's Guide. SGI Inc, 1996-1999. <http://www.sgi.com/tech/stl/>.

[12] Stroustrup, B. The Real Stroustrup Interview. IEEE Computer 31(6), June 1998, pp. 110-114.

Change-Merging of PSDL Abstract Data Types

David A. Dampier
Department of Computer Science
Mississippi State University
Mississippi State, MS 39762
(662) 325-8923
dampier@cs.msstate.edu

Vineet Chadha
Department of Computer Science
Mississippi State University
Mississippi State, MS 39762
(662) 325-8832
vineet@cs.msstate.edu

Abstract

Software development as an enterprise is at a critical place in history. We are now developing needs for software faster than we can develop solutions. The future of software engineering is in providing computer-aided tools for automating as much of the evolution process as possible, so we can hope to meet the software needs of the future. Our argument is that software should not be built and then maintained, but should be evolved, from the first requirements analysis, to system retirement. If we build software for evolution, it is generally easier to make changes to that software, whether they are corrective, adaptive, or perfective. This paper outlines a method of applying changes to abstract types that extends a previous change-merging method for executable prototypes written in the Prototype System Description Language (PSDL) [19]. PSDL prototypes consist of a set of operators and a set of abstract data types. Previous work focused on merging changes to PSDL prototypes consisting of only operators. Our current work is aimed at providing a model for merging changes to PSDL types as well, thus completing the change merging method for complete prototypes. This paper contains a model for change-merging PSDL abstract data types, as well as a consistency theorem that demonstrates the model's correctness.

KEYWORDS: Change Merging, Program Integration, Abstract Data Types, Classes, Software Evolution, Software Automation, Multiple Inheritance.

1. INTRODUCTION

Since the first computer program was written, software developers have been looking for innovations in software development. Some of the areas focused on are: programming languages, computer-aided tools, object-orientation, and process improvement. None of these innovations has been the elusive "silver bullet" that some are looking for [8]. Everyone agrees that there has to be a way to make software development better, but few can agree on how. What is commonly accepted today is a need for developers to focus on building software that can evolve.

Software systems are becoming increasingly huge and complex. In order to keep up with growing software needs and take advantage of increasing hardware capabilities, software must be designed, built, and delivered in less than twelve months. Additionally, this software must be evolvable, so when technology innovations occur, changes can be made more rapidly to the software to take advantage of these innovations. Research into computer-aided prototyping, where systems are built with executable specifications is showing some promise of providing a way to rapidly build software systems that more accurately satisfy the user's needs. However, all of the capabilities are not yet available to provide continuing evolution support to those same systems, once they are delivered. Incorporating changes into those systems requires reworking the original specification, and regenerating the executable code. Where this is often efficient, and allows maintenance of only the specification level code, it discounts the need for maintaining adaptive changes unique to particular environments.

Any software, modified after development, is very difficult to maintain. Changes made to any software system can be of three types: corrective, perfective or adaptive. Whenever an evolutionary change is made to the base version of a program, and the new version of the program is customized for a customer, perfective or adaptive changes made to the previous version of the software must be re-applied to the new version. This can produce inconsistencies in the existing version of the software. The answer is to build the software initially with the knowledge that it will change, and that the base version will evolve. In this way, the software can be built for evolution. To do this effectively, computer-aided tools are needed that will allow changes made to the base version of the system to be integrated automatically into each unique version. The first substantive work done in this area by Berzins in [2] provided models for understanding how two different enhancements made to the same software artifacts can be merged to produce a version with the characteristics of both enhancements. This pioneering work provided the basis for several follow on efforts in merging and integration. These follow on efforts include the work of Horwitz, Reps and others at the University of Wisconsin-Madison on integrating different versions of while programs [7, 18, 22] and Berzins and Dampier at the Naval Postgraduate School on merging changes to data flow programs in the form of PSDL prototypes. [3, 4, 5, 6, 14, 15]

Dampier did the first substantive work on change-merging PSDL programs [14, 15]. He constructed a model and method for automatically combining different versions of a prototype written in the PSDL. Prototypes written in PSDL consist of a set of PSDL data types and a set of PSDL operators. Dampier's work was limited to semantics-based change-merging of PSDL operators [14, 15]. The focus of this paper is the exploration of semantics-based change-merging of PSDL types. This is a part of the prototype change-merging problem that has not yet been explored. The next section will review the PSDL change-merging problem, and provide a context for the work outlined in this paper.

2. CHANGE-MERGING OF PSDL PROTOTYPES

Change merging is a process that allows different changes to a software product to be combined using computer-aided tools. Change merging can be done in two fundamental ways: semantics-based and syntax-based. Syntax-based change merging is performed on the source code of the input versions with respect to the differences in the syntax of each version. Semantics-based change merging is performed on the functions computed by the software product with respect to the behavior associated with each input version. Semantics-based change merging requires a solid mathematical foundation to provide some guarantee of the correctness and engender confidence in a working change-merging system [15].

PSDL is the language used to build prototypes in the Computer-Aided Prototyping System (CAPS) [20]. CAPS provides the designer with a set of computer-aided tools to quickly build a specification for a software system using PSDL, retrieve or build required primitive software components, and generate an executable prototype of the system. This prototype is then demonstrated to the customer, and based on the customer's comments, is updated to satisfy the customer's updated requirements. A graphic of a prototyping paradigm similar to that used in CAPS is shown in Figure 1.

PSDL prototypes are constructed as sets of operators and data types, where each of the components can be either composite (constructed from collections of other PSDL components) or atomic (implemented in a high level programming language, like Ada or C++). The operators implement either functions or state machines and the PSDL types are abstract data types (ADTs) containing both data and methods to operate on those data.

During construction of a sufficiently large prototype, it may be necessary to distribute different pieces of the design effort to different designers, and ultimately integrate their individual efforts into a cohesive prototype. This integration effort generated a need to provide automated tools that would provide the capability to integrate these individual efforts in a safe way without the need for extensive human intervention. Our initial efforts on this problem were very successful. [4, 14, 15] Unfortunately, our efforts until now have been limited to merging changes to prototypes with only operators, without regard to the data types that will doubtless be required in large prototypes. In [13], an approximate method for merging changes made to PSDL operators was provided. In [14, 15], a slicing method was provided that used the approximate method for completing the change-merge, but validated the results through the use of prototype slicing.

The existing model and method for change-merging PSDL prototypes consists of a model for change-merging operator specifications and a method of slicing prototype implementation graphs that provides a mechanism for guaranteeing semantic correctness of a syntactic merge of the implementation graphs. The basis for this guarantee is the Slicing Theorem that states that the behavior of any slice of a prototype will remain precisely the same in any prototype, as long as the slice is precisely the same. [15] The benefit of this theorem is that we can show that as long as the behavior of a slice with respect to an affected part of a modified prototype remains the same in the change-merged version, then the significant change in that modification is preserved through the change-merge operation. The rest of the model uses lattice theory, along with Boolean and Browerian algebras to construct the change-merged specifications. The focus of the rest of this paper is on merging changes to PSDL types.

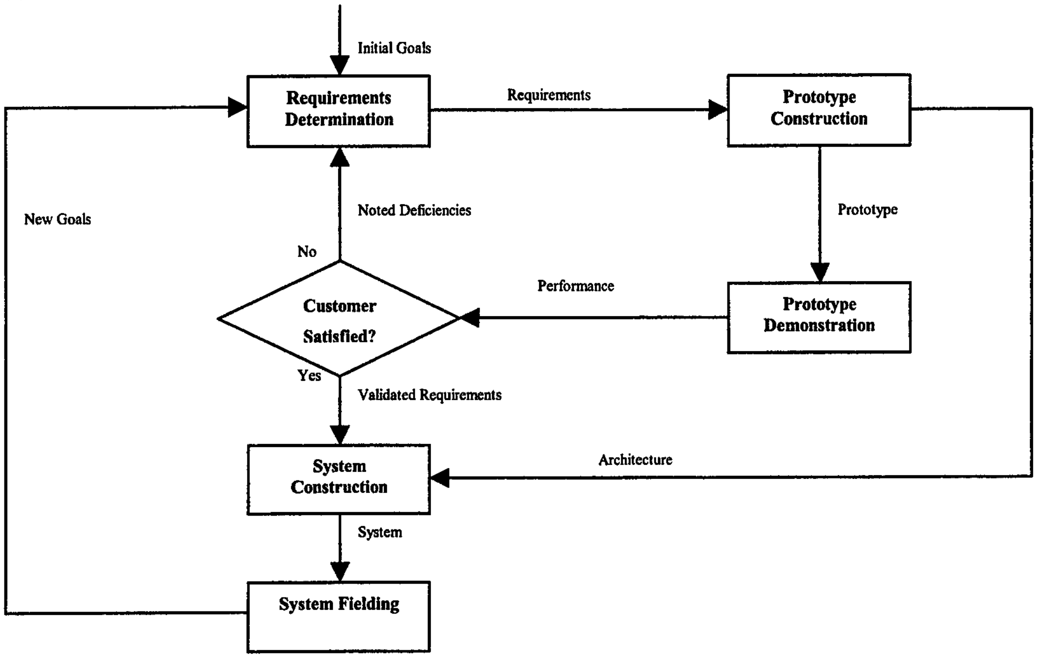


Figure 1: Rapid Prototyping in CAPS [15]

3. PSDL TYPES

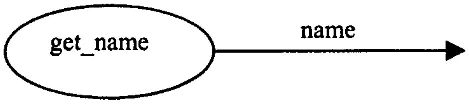
PSDL prototypes consist of both types and operators. PSDL types are abstract data types. Their basic structure is the same as any abstract data type, with a set of attributes and a set of methods. All declared types in PSDL prototypes are PSDL types. The structure of the type specification for these types is the same, whether the implementation is in PSDL or some high-level implementation language. As already stated, PSDL types consist of both data and operators to affect those data. The basic structure of a PSDL type contains both specification and an implementation. The specification consists of a set of attribute definitions, as well as specifications for the methods or operators to operate on those attributes. An example of a PSDL Type specification is shown in Figure 2. PSDL keywords are shown in **boldface**. This is the specification for a basic class of *Person* containing simple data like their name.

The implementation part of the PSDL type, as was stated earlier, can either be in PSDL or in some high level language like Ada or C++. For simplicity, and because our current version of CAPS uses Ada exclusively, all examples of non-PSDL implementations will be shown in Ada. The alternate structures of the implementation part of the PSDL type for both PSDL implementations and Ada implementations are shown in Figure 3. Again, PSDL keywords are shown in **boldface** text. Even in a PSDL implementation, some operators may be implemented in a high-level language instead of PSDL. This mixed implementation structure may seem complicated, but as you will see, does not affect our change-merging method at all.

```
Type Person
    specification
        name: string

        operator create
            specification
                input name: string
            end

        operator get_name
            specification
                output name: string
            end

    end
```

Figure 2: Example of a PSDL Type Specification for *Person*

```
Type Person
    specification

          •••
      end

    implementation ada Person_Package end

             or

Type Person
    specification

          •••
      end

    implementation Person

        operator create
            implementation ada create_person end

        operator get_name
            implementation
```



```
            end

    end
```

Figure 3: Example of different PSDL Type Implementations for *Person*

4. CHANGE MERGING OF PSDL TYPES

We started our research by searching for accomplished work on merging changes to abstract data types. Finding none, we took the view that the merging of changes made to abstract data types is similar to multiple inheritance. As it turns out, there is considerable research to support this notion. We will start our look at change-merging PSDL types with a look at some of this research.

4.1 Inheritance Models

In [10], the author discussed efficient handling of class hierarchies. He argues that a lattice structure was suitable for class hierarchies for the following reasons:

- Lattices allow resolution of multiple inheritance conflicts exactly. They also help to reduce the complexity of implementing conflict resolution strategies.
- Lattices help in writing typing inference algorithms.
- It is easier to represent large complex software systems in terms of small lattice structures.
- Lattices support compact encoding techniques.

Figure 4 shows an inheritance lattice from [10]. It is easy to see how the multiple inheritance from both Student and Employee could cause problems in StudentEmployee. One example of a conflict is in the department to which each of the persons is assigned. Certainly students are assigned to a department that they are studying in, and Employee are assigned to a department in which they work, but what happens when an employee in the Math department is a student in the Computer Science department. This is an example of the kind of conflicts that can be caused by this kind of inheritance.

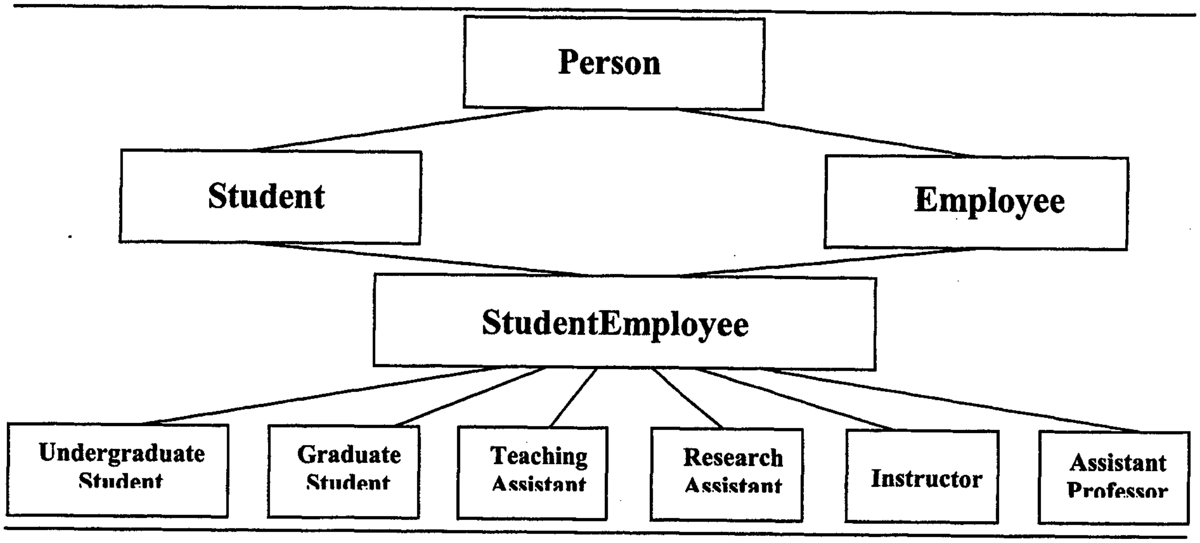


Figure 4: An Inheritance Lattice [10].

In [12], the author described inheritance as a mechanism for incremental programming. Based on Cook's inheritance mechanism, Benattou and Lakhal presented an incremental formal model of both single and multiple inheritance [1]. Their model allows a new class to be defined by incremental modification of existing classes. The authors also address the automatic conflict solving in multiple inheritance using two operators: $\oplus c$ a combination operator on structures with conflict resolution, and Δc a multiple inheritance operator. Their paper addressed name and value conflicts and possible ways of solving those conflicts. They suggested solving name conflicts by explicit designation and value conflicts by a process called *linearizing*. Ducournau et al also addressed conflict resolution mechanisms for inheritance. According to them, there can be two possible kinds of conflicts: value conflicts, and name conflicts. Value conflicts are due to having different values for the same attribute. Name conflicts are due to different attributes having the same name [16].

In [9], the authors addressed conflicts between methods of the same name in superclasses. Consider the following example:

Student
Superclass: Person
Methods: CardNumber()
ValidateCard()

Employee
Superclass: Person
Methods: CardNumber()
ValidateCard()

Consider these two as base classes in a class hierarchy diagram. We can get a *StudentEmployee* as an inherited class, much in the same way as show in Figure 4. According to the author, a class must inherit all of the characteristics of their superclasses. The authors considered two kinds of strategies for solving conflicts: linear strategies and graph-oriented strategies.

4.1.1 Linear Strategies:

Languages such as *CommonLoops* follow linear strategies. These strategies are based on a common principle, "flatten the inheritance graph to a linear chain without duplicates, and then treat the results as single inheritance" [9]. It converts the partial ordering of classes into a total one by using relative ordering of classes within the list of the direct superclasses of each class. These strategies oppose the tenets of object-orientation such as reusability, incremental design and modularity.

4.1.2 Graph-Oriented Strategies:

Graph-oriented strategies deal with complete hierarchical class diagrams. When a conflict arises, they can specify the superclass from which they wish to inherit. In [17], the authors proposed a formal method that produces a lattice structure called a Galois lattice from a given set of classes. This method of building lattice structures from a given set of classes has the following advantages:

- It supports an efficient incremental update algorithm.
- It does not depend on input ordering.

In [21], the authors consider an approach for the extension and merging of a base system in a library or existing application. The authors suggested that successive extensions can be combined using an *extension* operator and parallel extensions can be combined using a *merge* operator. Conflicts should be resolved in the merge operation.

4.2 Possible Conflicts in Abstract Data Type Merging and the methods for Resolving Them

Based on our research, we have identified several types of conflicts that have to be resolved when merging two PSDL ADTs. The different conflicts fall into the following basic categories: types, attributes, and methods.

4.2.1 Type Set Merging

Type Set conflicts arise when two prototypes are being merged that contain new types with different names, or have removed types from the base still used by the other modification. These conflicts can occur when two independent changes are made to the same prototype, where new type definitions are required for the change, or old type definitions are no longer required. Since all PSDL prototypes contain a set of data types, the correct solution to this problem can be found through the use of a Powerset lattice, or Boolean algebra. Both modified versions of the prototype contain a different set of types than the base version of the prototype. The minimal merged version of the set of types can be constructed through the use of the following change-merge equation:

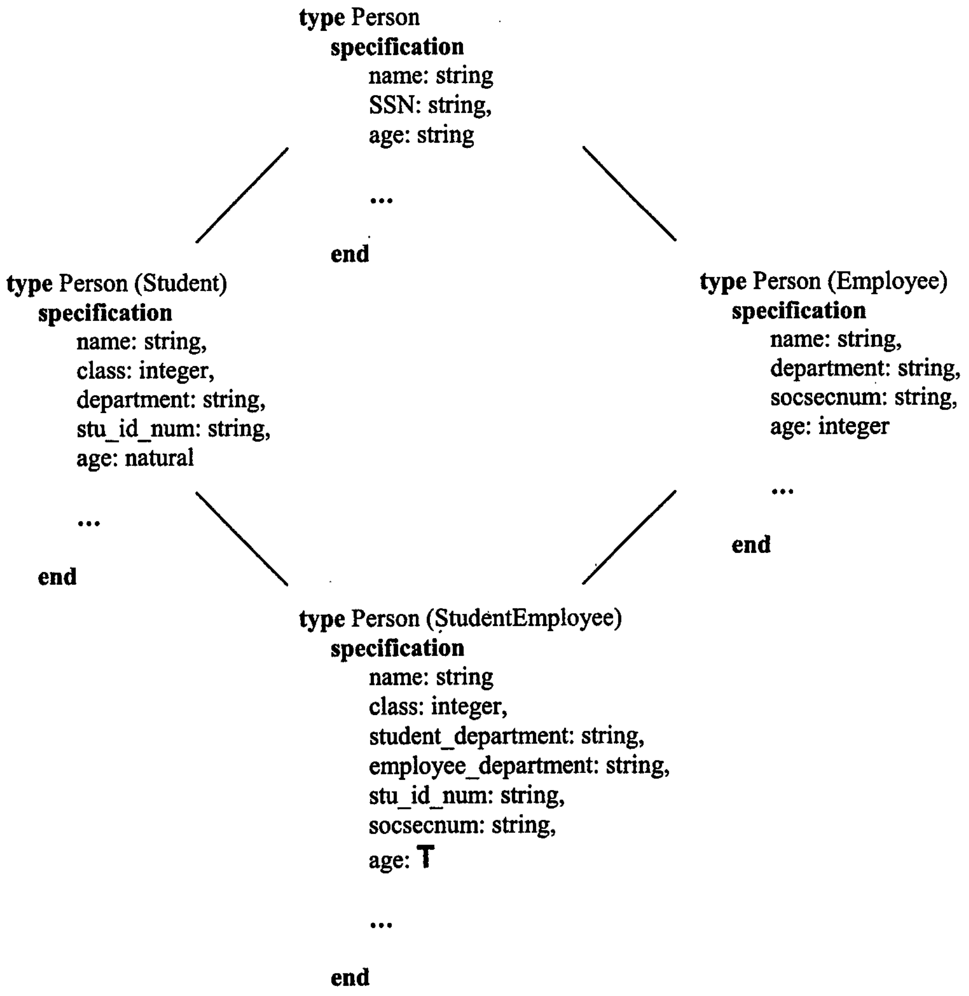
$$\text{Merge} = (A - \text{Base}) \cup (A \cap B) \cup (B - \text{Base}) \text{ [3]}$$

This equation chooses the types that have been added in A and B, along with the types that have been preserved from the Base version in both A and B. The only possible problem that could occur as a result of this change-merge, is if a type that has been removed in one modification is needed for a new method or attribute in the other modification. In this case, that type can be added back into the prototype as needed. This set of types can contain more than the minimal set without conflict. Additionally, each of the modified versions can contain a new type by the same name, but with different specifications and implementations. In this case, the solution is simple, these two types should be renamed, perhaps by adding the version number of the modification, and treated appropriately as different types.

4.2.2 Merging Attributes

Now, we consider the case where different changes have been made to the same PSDL type in each of the modified versions. At the top level, the set of attributes in the merged version of the type can be combined using much the same method as described in the previous section for the sets of types in a prototype. Obviously, if an attribute is removed in the first modification, and not in the second, then its removal was significant to the designer of the first modification, so that change should be preserved. Since any method that operated on that attribute would also have had to be removed in the first modification, its removal would also be significant and must be preserved in the merged version. Any use of that type by a new method in the second modification would cause a conflict, but this is a conflict that would have to be resolved by the designer.

Similarly, added attributes in each of the modified versions of the PSDL type would be included in the merged version. Let us look at some examples of these conflicts and their resolutions (See Figure 5), borrowing from the Student and Employee problem seen earlier.



```
type Person
  specification
     name: string
     SSN: string,
     age: string

     ...

  end
```

```
type Person (Student)
  specification
     name: string,
     class: integer,
     department: string,
     stu_id_num: string,
     age: natural

     ...

  end
```

```
type Person (Employee)
  specification
     name: string,
     department: string,
     socsecnum: string,
     age: integer

     ...

  end
```

```
type Person (StudentEmployee)
  specification
     name: string
     class: integer,
     student_department: string,
     employee_department: string,
     stu_id_num: string,
     socsecnum: string,
     age: ⊤

     ...

  end
```

Figure 5: An Example of Attribute Merging

As can be seen from this example, in the case where new attributes were added in each of the modified versions, those new attributes appear unchanged in the merged version (stu_id_num, socsecnum, class). In the case where attributes in the base version were removed in one or both of the modified versions, it does not appear in the merged version. In the case where new attributes were added to both of the modified versions with the same name, but with different meanings, they were both renamed and included in the merged version. This example does bring to light a possible problem we have not yet discussed. Age appears in all three input versions, but with three different type declarations: **string**, **integer**, and **natural**. Rightly, we have included the definition of the age attribute in the merged version, but with the type identifying a conflict. This is the safest way to include the attribute, but it may not be the only way. Consider this particular example. In the Student version of the ADT, age is defined as a **natural** number, and in the Employee version, it is defined as an **integer**. Since both of these are different from the Base version, it is indeed a conflict, but since the set of **naturals** is a subtype of the set of **integers**, it may be possible to take the most restrictive version and resolve the conflict automatically. We are still exploring this possibility and hope to have an extension to this model soon.

4.2.3 Merging of Methods

For merging the sets of methods, a Boolean algebra based merging operation similar to those described in the two previous sections is also applicable. However, in addition to the merging of sets by name, we have some additional concerns with methods. First, let us say that in the case where we have three different versions of the same method being included where the three methods are PSDL operators, previous results from [4, 14, 15] can be applied to effectively produce the merged version of the operator. Likewise, if the three implementations are Ada implementations (or some other high level programming language), we must defer to an as yet undiscovered method for merging changes to operators in those·languages. The following cases refer to different problems we have been able to enumerate in our study of the ADT merging problem.

4.2.3.1 Return Parameters are Different

When the methods are the same, and the return parameters are different in one of the modified versions, include the modified version in the merged type. If the return parameters are different in all three versions, there is only one feasible option, report a conflict. If we try to choose between the two modifications, we may produce a version that is not safe.

4.2.3.2 Arguments are Different

When the methods are the same, but the input arguments are different, there are three possible options:

- Include one of the methods: this is not safe unless the method in one of the modified versions is the same as the base version, as it doesn't take into consideration why they are different.
- Report a Conflict: this is maximally safe, but provides no benefit other than safety.
- Include both of the methods: this too is safe, but is similar to function overloading available in most object-oriented programming languages.

4.2.3.3 Arguments are the Same but Listed in Different Order

When the methods are the same, but the input arguments appear in a different order, we have the same three options as if the arguments are different. The most appropriate choice for resolving this conflict is to include both of the modified versions of the method.

5. CONSISTENCY THEOREM

In [11], we provide a model and method for merging abstract data types in the general case. Mathematical equations are also provided to describe each of the above operations on generic abstract data types. Also in [11], a theorem is provided that shows that the result of merging two different modifications of a base abstract data type results in a set of attributes and methods which are consistent. This theorem is based on the following definitions:

S_A is the set of attributes in an abstract data type, A.

M_A is the set of methods in an abstract data type, A.

P_A is the set of input parameters to the methods contained in M_A.

$\gamma(S_{A,} M_A)$ is a consistency predicate that is TRUE if and only if the methods contained in M_A are consistent with the attributes contained in S_A . This consistency is shown by the relation $P_A \subseteq S_A$. Based on this consistency predicate, γ, the following theorem is stated: (The proof of this theorem is provided in [11])

Consistency Theorem

If Base, A, and B are three versions of an Abstract Data Type, and

$$\gamma(S_{Base,} M_{Base}) \quad \Leftrightarrow \quad P_{Base} \subseteq S_{Base}$$

$$\gamma(S_{A,} M_A) \quad \Leftrightarrow \quad P_A \subseteq S_A$$

$$\gamma(S_{B,} M_B \quad \Leftrightarrow \quad P_B \subseteq S_B$$

then $\gamma(S_{A[Base]B,} M_{A[Base]B})$.

6. CONCLUSION

Change merging of abstract data types in general is applicable to many software evolution activities. It also has some potential in the areas of software reuse and reengineering. Today, due to the increased complexity of software systems, a primary focus is component-based development and reusable software. Abstract data type change merging can help in the integration of two concurrent developments of an object or the automatic integration of two or more changed versions with respect to the base version they are created from. We have explored a basic change-merging problem as it applies to the prototype-merging problem and have described our results. This is a complicated research problem, and although the results described in this paper are very encouraging, much work in still necessary to provide a working change-merging tool for PSDL.

References

[1] Benattou, M., and Lakhal, L., "Incremental Inheritance Mechanism and its Message Evaluation Method", *Proceedings of 3rd Basque International Workshop on Information Technology*, July 1997, pp. 159-168.

[2] Berzins, V., "On Merging Software Extensions", *Acta Informatica* 23, pp. 607-619.

[3] Berzins, V., "Software Merge: Semantics of Combining Changes to Programs", *ACM Transactions on Programming Languages* 16(4), pp.1875-1903.

[4] Berzins, V. and Dampier, D., Software Merge: Combining Changes to Decompositions. *Journal of Systems Integration* 6, 1996, pp. 135-150.

[5] Berzins, V., and Luqi, *Software Engineering with Abstractions,* Addison-Wesley Publishing, 1996.

[6] Berzins, V., "Merging Changes to Software Specifications", *Lecture Notes in Computer Science*, October 1997, pp.121-131.

[7] Binkley, D., Horwitz, S. and Reps, T., "Program Integration for Languages with Procedure Calls", *ACM Transactions on Software Engineering and Methodology*, ACM Press, January 1995.

[8] Brooks, F., "The Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, 20(4), April 1987, pp. 10-19.

[9] Carré, B. and Geib, J., "The Point of View Notion for Multiple Inheritance", *Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1990, pp. 312-321.

[10] Caseau, Y., "Efficient Handling of Multiple Inheritance Hierarchies", *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, And Applications*, September 1993, pp. 271 – 287.

[11] Chadha, V., *Automated Evolution of Abstract Data Types*, Master's Thesis, Mississippi State University, Mississippi State, MS, September 2001.

[12] Cook, W., "Interfaces and Specifications for the Smalltalk-80 Collection Classes", *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, And Applications*, October 1992, pp. 1-15.

[13] Dampier, D., *A Model for Merging Different Versions of a PSDL Program*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[14] Dampier, D., Luqi, and Berzins, V., "Automated Merging of Software Prototypes", *Journal of Systems Integration,* 4(1), pp. 33-49.

[15] Dampier, D., *A Formal Method for Semantics-Based Change Merging of Software Prototypes*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1994.

[16] Ducournau, R., Habib, M., Huchard, M., and Mugnier, M., "Monotonic Conflict Resolution Mechanisms for Inheritance", *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, And Applications*, October 1992, pp. 16-24.

[17] Godin, R., and Mili, H., "Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices", *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, And Applications*, September 1993, pp. 394-410.

[18] Horwitz, S., Prins, J., and Reps, T., "Integrating Non-Interfering Versions of Programs", *ACM Transactions on Programming Languages and Systems* 11(3), pp. 345-387.

[19] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", *IEEE Transactions on Software Engineering* 14(10), October 1988, pp. 1409-1423.

[20] Luqi, "Software Evolution through Rapid Prototyping", *IEEE Computer*, May 1989.

[21] Ossher, H., and Harrison, W., "Combination of Inheritance Hierarchies", *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, And Applications*, October 1992, pp. 22-40.

[22] Ramalingam, G., and Reps, T., "A Theory of Program Modifications", *Lecture Notes in Computer Science* 494, pp. 137-152.

Formal verification of embedded distributed systems in a prototyping approach

F. Kordon, I. Mounier, E. Paviot-Adet
LIP6-SRC, Université P.&M. Curie
4 place Jussieu, 75252 Paris Cedex 05, France
Fabrice.Kordon@lip6.fr
Isabelle.Mounier@lip6.fr
Emmanuel.Paviot-Adet@lip6.fr

&

D. Regep
PhD. student CS TELECOM
28, rue de la Redoute, BP 74
92263 Fontenay-aux-Roses Cedex, France
Dan.Regep@lip6.fr

Abstract: *This paper presents an evolutionary prototyping methodology dedicated to the design, verification and implementation of embedded systems. This methodology relies on **LfP**: a formalism combining UML-like structuring capabilities and a precise semantic suitable for both code generation and formal verification based on colored Petri nets. We apply this methodology on a small example and show how it enables system designers to detect non-trivial problems on the system.*

Keywords: *Prototyping, Formal verification, Petri Nets, UML.*

1. INTRODUCTION

Design and implementation of industrial systems is getting more and more complex **[9]**. This is a problem for embedded distributed systems for which a high quality is required. Several problems can be identified:

- Standard notation, such as UML **[15]** can be considered as an important contribution to describe a solution. However, it is more suitable at an early stage of application design and implementation. Thus, UML specifications are difficult to check due to their semi-formal semantics (dynamic aspects are not formally defined). A typical illustration is the interaction between components of a system. This information is dispatched into several diagrams: interaction, sequence and state.
- Once the specification of the system is completed, implementation need to be done. Then, developers may interpret these specifications and we can get a program which is not exactly the image of the corresponding specification.
- Tests of the system are usually performed on the program. Then, when debugging the program, the initial specification tends to disappear: modifications on the program are not reported to the specification. Security of such modification usually decreases in the maintenance phase of the system.

Evolutionary prototyping **[13]** is a good solution to these problems since it enhance the definition of a model serving as a basis for both the description of the system and automatic code generation. By reducing the production cost of an executable program, it promotes the model to be the center of the development process. Then, a system is elaborated by successive refinements of the following operations:

- design/refinement of the model,
- evaluation of the model,
- code generation,
- evaluation of the prototype,

In evolutionary prototyping, a strong correspondence between the model, programs and documentation may be maintained. However, evaluation of the system still rely on «traditional» testing techniques based on large benchmarks.

This paper presents an evolutionary prototyping technique. Our methodology relies on **LfP** (Language for Prototyping), a formalism dedicated to the description of embedded distributed systems **[18]**. **LfP** combines high level modeling facilities such as the one of UML and a precise semantics suitable for both code generation and system verification by means of formal methods (instead of benchmarks-based testing).

Section 2. presents our prototyping methodology. Then, **LfP** is described in Section 3. Section 4. details an example of system specification using **LfP** and states some properties to be checked on this system. Finally, Section 5. shows how a formal specification can be generated from the **LfP** model and used to detect non-trivial errors.

2. METHODOLOGY

Our methodology is a model-based development in the sense of **[17]**: the model describes the system and serves as a basis for validation (in our case, formal verification) and code generation. Our methodological approach aims to implement evolutionary prototyping capabilities based on:

- *An integrative design approach.* **LfP** acts as a glue prototyping language **[2]** between state of the art specification formalisms (e.g. UML for system modeling, ODP as a distributed component framework **[8]**, Petri nets for formal verification).
- *An aspect oriented design framework.* **LfP** is based on a multi views approach to system prototyping **[7]**. Views are dedicated to a given prototyping aspect: software architecture, system implementation and formal property description.
- *A formalized development approach* to system behavior modeling and verification **[19]**. **LfP** relies on well formed Petri nets semantics **[3]** for formal verification.
- *A hierarchical, structured and modular approach* to system modeling **[4]**. **LfP** uses a component based approach allowing hierarchical specification and behavior refinement.

The main objective of **LfP** is to formalize relations between system modelling, formal verification and code generation of embedded distributed systems. Thus, we provide:

- transparent formal verification to enable its use in an industrial context without requiring specific training and skills **[12]**,
- strong correspondences between the detailed descrip-

tion of a system, its proofs and its implementation. In other words: «what you check and what you implement is what you describe».

Figure 1 presents our methodology. It takes in input an UML specification of a system. UML is not suitable for direct verification as noticed in the vUML project [10]. This is also true for distributed code generation from UML as mentioned in [14]. So, extensions to UML have to be considered. LfP has been elaborated for this purpose and can be seen as an additional UML diagram.

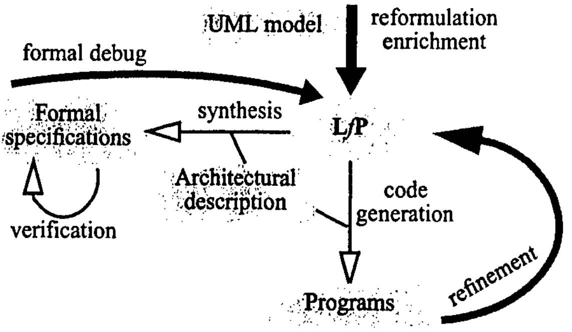


Figure 1 : Our evolutionary prototyping methodology.

The **LfP** model can be partially generated from UML standard diagrams. However, it contains enriched information compared to other UML diagrams: centralized description of components behavior by means of finite state machines (FSM), identification of properties to be verified and implementation directives. All this information is located in **LfP** and can be checked without being concerned with coherence problems between several diagrams.

Once the **LfP** model is produced, Petri nets synthesis can be performed. In Figure 1, «synthesis» corresponds to a set of transformations from **LfP** to Petri nets. Each one is dedicated to the verification of a given property according to a given strategy. This reduces the complexity of the proof: non relevant information can be discarded and thus, generated Petri nets are optimized.

Once all properties stated in the **LfP** model are verified (which may require some modification and several refinement son the diagram itself), code generation produces pieces of programs to be compiled and deployed in the target execution environment.

As shown in Figure 1, the main interest of evolutionary prototyping is to enhance the role of a model which enables: 1) several refinement of the system since production of the corresponding executable version is performed at low cost, 2) formal verification manageable by engineers since most of the process is hidden and performed automatically, 3) use of the **LfP** model, even during the maintenance phase.

To operate our methodology, we use a set of languages dedicated to each prototyping phase:

- UML for system specification and modelling,
- **LfP** diagram to centralize informations and to enable code generation as well as formal verification,
- Petri nets to apply formal verification procedures,
- Programming languages to implement the system.

3. THE LFP FORMALISM

This section summarizes the main features of **LfP**. Detailed information and rationale can be found in [18].

LfP is a graphical Architecture Description Language with coordination facilities. It is dedicated to the rapid prototyping of embedded concurrent systems. **LfP** enhances an existing UML model with information enabling automatic code generation of concurrent programs and formal verification.

To do so, **LfP** uses three orthogonal views adapted to some specification aspects:

- the *functional view* (implemented as a diagram),
- the *implementation view* (textual annotations on the diagram),
- the *property view* (textual annotations on the diagram).

The functional view describes the system behavior in terms of execution workflow of connected components and the coordination between component instances. It also describes the system software architecture.

The implementation view describes the system implementation constraints (target executive, used programming language, communication infrastructure) and the deployment topology.

The property view specifies properties to be verified by the system (similar to the B proof-assertions [1]). Such properties are stated by means of invariants (for example, to check mutual exclusion), temporal logic formulas (for example, to check availability or fairness of a service) or other statements that can be converted to a given formal method. This view can be exploited to perform computer-assisted formal verification but also introduces relevant information for code generation (i.e. rutime checks).

3.1. The LfP Structure

The **LfP** functional diagram contains:

- a declarative part defining management information (e.g. model name, author, version number, comments and the associated UML model if any) and formal declarations: types or constants. Elementary types are: integer range, ordered enumerations or the opaque type. The opaque type denotes variables which only support the affectation operation and, thus, cannot influence the execution workflow.
- a list of entities: classes and media.

A **LfP** class corresponds to a complete UML instanciable class. Thus, abstract or virtual UML classes have no correspondence in **LfP**.

A media is used to connect classes. It specifies both interaction contract and communication semantics. It corresponds to an UML association, aggregation or composition.

Table 1 presents the graphical representation for classes and media.

LfP Class	**LfP Media**
Class name	Media name

***Table 1:** Graphical representation of classes and media*

3.2. LfP entities

As mentioned in Section 3.1., the **LfP** functional diagram contains classes and media. Their description strongly relies on **LfP**-FSM (Finite State Machine) supporting various elements of a class or media specification. Thus, we present the L/P-FSM structure prior to classes and media.

3.2.1. L/P-FSM

L/P-FSM uses a notation similar to the one of Petri nets and provides some modelling facilities. They are used in various parts of a specification; the main difference consists in the signification of transition labels. L/P-FSM contains:

- a declarative part specifying a list of variables representing the execution context.
- the FSM itself. It specifies the execution workflow of a class, a class role, a method or a media. A L/P-FSM contains basic elements (or nodes) «wired» together using connection links.

Variables of a L/P-FSM context are either local to a class or media instance or shared between all of them. Variables are typed (according to a visible defined type) and may hold a default value. As mentioned in Section 3.1., opaque variables only support the affectation operation.

Symbol name	Icon
State	○ ◎BEGIN ●FINAL
Transition	LOOP ← transition name □ `<i <= 3>` ← guard condition `i++;` ← statement `[i > 0]` ← safety condition
S-Transition	■LOOP ← transition name alias
H-Transition	□block ← sub-net name
Barrier	SYNC ← barrier name
Protector	variable.lock ← protector name (4) ← protector cardinality
Binder	SC_channel.server ↖ reference to a binder binder name → server
Constructor	ConveyerControler ← target class name

Table 2: Graphical representation of L/P-FSM basic elements

Table 2 summarizes nodes to be found in an L/P-FSM:

- *States* are execution steps. Two special states are distinguished: BEGIN and FINAL corresponding to the initial and final execution states. L/P-FSM has only one initial state.
- *Transitions* express potentially guarded actions. Guard conditions specify activation rules to be satisfied when firing a transition. The transition name may reference class role name (Section 3.2.2.), or a class method name. A statement is executed after the firing, it modifies state variables of the L/P-FSM visible at this level. These have an atomic execution semantics. Safety conditions express invariants useful for formal verification, debugging and testing
Transitions may be linked to sequential code written using any programming language, to be inserted in the distributed application at code generation time. This code may change opaque variables values only and thus, cannot change the execution workflow.
- *Shadow Transitions* (S-Transitions) are graphical aliases to existing transitions proposed to simplify L/P-FSM.
- *Hierarchical Transitions* (H-Transitions) abstract sub-L/P-FSM to increase readability. Sub-L/P-FSM have one initial state and one terminal state. These are bound to the H-Transition input state and output state.
- *Barriers* are special shared transitions corresponding to a synchronization point between all concurrent instances of a L/P-FSM.
- *Protectors* are shared locks (multi-level semaphores or groups of semaphores) used to provide restricted access to a shared resource. They are used to define critical sections between concurrent instances of a L/P-FSM. A protector can be standalone or associated to one variable or group of variables. The protector cardinality specifies how many concurrent L/P-FSM instances may simultaneously get into the critical section.
- *Binders* are access points to media. L/P-FSM communicate through binders by means of messages. A message consists of three fields: 1) a message name known by the media, 2) message discriminants that can be modified by the media, 3) message arguments that must be opaque for the media. Binders are declared in media and referenced in classes.
- *Constructors* are used to create new class instances. An initialization context has to be specified for created instances.

Table 3 presents connectors to be used in a L/P-FSM.

Arc	Protector link	Media link
⟶	──◇	──↠ ↞── ↞──↠

Table 3: Graphical representation of L/P-FSM connectors

- *Arcs* are used to link a State to a Transition, S-Transitions, H-Transition or Barrier and vice versa. An arc express the execution sequence. L/P-FSM are sequential finite state machines. Thus, the number of input and output arcs of a Transition (S-Transition, H-Transition, or Barrier) is of exactly one.
- *Protector Links* connect Protector to Transitions or S-Transitions and vice versa to define critical sections.
- *Media Links* are used to connect Binders or Constructors with Transitions or S-Transitions. Media Links specify the connection direction (in, out or inout). A Media Link specifies the binding contract between local context variables and messages (discriminant, name and arguments).

3.2.2. L/P Classes

A L/P Class corresponds to an UML implementation class and expresses some functional aspects of a system. It consists of:

- a declarative part specifying: 1) the class identifier, 2) for each binder, potential messages and their parameters (some of these messages correspond to public methods), 3) a list of private methods and their parameters, 4) defintion of sequential code to be linked to transitions.
- a list of FSM defining : 1) the execution contract (main FSM), 2) class roles (optional), 3) methods.

Definition of the main FSM is mandatory. It represents the execution workflow of a class instance. Transitions in the main FSM may reference class roles (if any) or class methods.

Class roles correspond to alternative class behaviors; their definition is optional. Each role is described using a **L/P**-FSM. Transitions in a role may reference methods.

Class methods are also described by means of a **L/P**-FSM defining the execution workflow (i.e. the method execution contract).

Table 2 summarizes the graphical representation of the class main **L/P**-FSM, a class role or of a class method.

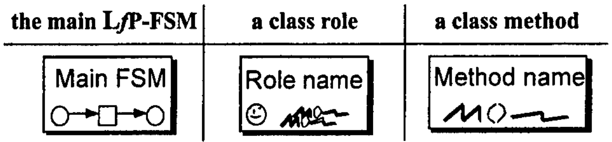


Figure 2 : Graphical representation of class components.

3.2.3. L/P Media

Media connect instances of **L/P** Classes. It is possible to use them as basic components or to assemble them into more sophisticated communication patterns. Media connecting two or more **L/P** classes correspond to an UML association, aggregation or composition. Media can also be used to implement shared resources (list, FIFO, stack, etc.).

A media specifies binding constraints and communication protocol semantics. We base our approach on the ODP contract definition **[8]**. A media consists of:

- a declarative part defining : 1) new types declaration 2) media variables which are similar to class variables, 3) the interaction contract consisting of several binding constraints.
- the main FSM representing the communication contract (communication protocol semantics).

The binding constraints specify: 1) a reference to the connected binding point, 2) the communication mode (synchronous or asynchronous), 3) the accepted messages and their arguments, 4) the binding multiplicity (one, `all` or any): one means that the binder is connected to only one class or media instance; `all` specifies that the binder is shared by all the connected classes instances; any leave this unspecified.

A media cannot play various roles, has no methods nor associated constructors. Media carry on information on classes request.

4. AN EXAMPLE

Let us consider a set of conveyers circulating on a path divided in N segments as shown in Figure 3. A segment may contain only one conveyer. Conveyers may cross between segments where a crossing zone is defined (noted Z in the Figure). When two conveyer cross, the first one get into a special path in the crossing zone and let the other one get out before entering in the segment.

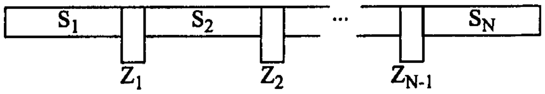


Figure 3 : The conveyer system.

4.1. Conveyer Behavior in the System

Conveyers, segments and crossing zones are locally driven by an embedded Control application. Command centers drive conveyers movements (using the MOVE message). However, these are not part of the system but correspond to an «external» component (e.g. a piece of code that already exists and has to be linked to the generated programs).

Therefore, the system contains three classes: SegmentControl (noted SC), ConveyerControl (noted CC) and CrossingZoneControl (noted CZC). Interactions between classes are defined using the following rules:

1) Upon receiving a MOVE command, a CC has to require (using DEM message) an authorization provided by the SC of the segment it wants to get in (if different from the current one). When it gets a positive answer (AUT message), it may come in.
2) This authorization may be refused (REF message), then, the conveyer must get into the crossing zone.
3) A conveyer stopped in a crossing zone is not considered to be in any segment.
4) The SC replies AUT when it is empty.
5) The SC replies REF when it contains a conveyer. It then store the query in a local FIFO to reactivate the demanding conveyer when it is empty.
6) The CC sends DEM when it wants to leave a segment, just before entering in a new one.
7) The CC leaves the crossing zone when it gets a GO message from the SC. This message is sent when the conveyer occupying the segment leaves it.
8) When a CC leaves a segment (to get into another one or to get into a crossing zone), it notifies the corresponding SC by means of a OUT message. This message is also sent when a CC leaves a CZC.
9) To increase security, CC checks (message EMPTY) if a CZC is empty when entrance in a segment is refused.
10) The CZC answers to EMPTY using OK (it is empty) or PB (it already contains a conveyer).
11) When PB is sent by a CZC, the CC sends ALARM to other conveyers and the entire system stops in an error state.

Figure 4 presents the static structure of the system as an UML class diagram.

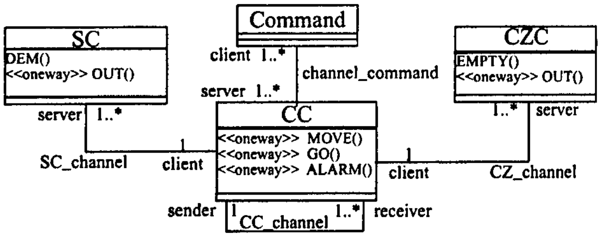


Figure 4 : The UML class diagram of the conveyer system.

Let us illustrate the rules exposed in Section 4.1. with UML sequence diagrams. The one of Figure 5 corresponds to a first scenario. Conveyer «c» located in segment «1» wants to get into segment «2». It sends DEM and gets AUT, enters in segment «2» and sends OUT to segment «1».

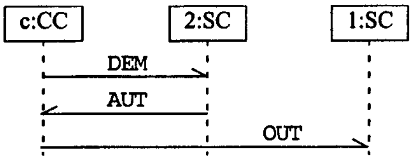


Figure 5 : Sequence diagram of scenario 1.

The UML sequence diagram of Figure 6 corresponds to a second scenario. A conveyer «c1», located in segment «1», wants to get into segment «2» where another conveyer «c2» is located. When «2» refuses entrance, «c1» gets into the crossing zone «z» after having checked if it is empty. When «c2» leaves «2», «c1» is waken up by «2» and leaves «z».

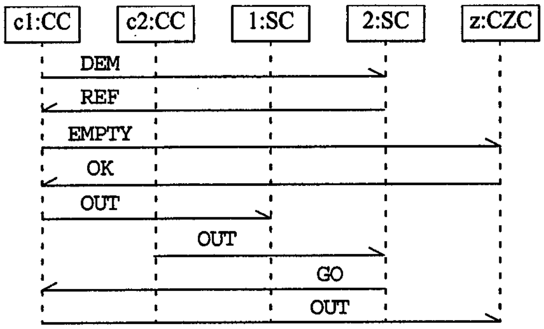


Figure 6 : Sequence diagram of scenario 2.

4.2. The **LfP** Specification

In order to build the **LfP** diagrams we reuse information found in the UML models.

4.2.1. Description of the System

Figure 7 presents the main **LfP** diagram. The static structure of this is derived from the UML class diagram (Figure 4). It declares three classes (corresponding to the UML ones) and four media. The first three media correspond to the three UML associations and specify the communication protocol between classes as well as their interaction contract with media. The last media is a local FIFO used by SC instances to store unsatisfied queries.

```
  model_name := 'conveyeurs' ;
  author := 'Dan Regep';
  version := 0.0.1;
  comments := '';
  UML_model := 'conveyers.mdl';

  const NBC := 2;
  const NBS := 4;
  const NBZ := 3;
  type station is integer range 1..NBS;
  type start_bound is integer (1,2,3);
  type end_bound is integer (1,2,3);
 static instances:
  CC with HID in range 1.. NBC;
  SC with HID in range 1..NBS ;
  CZC with HID in range 1..NBZ;
 properties:
```

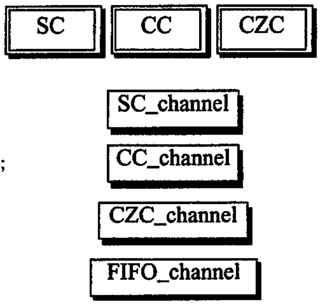


Figure 7 : The L*fP main diagram of the conveyer example.*

The declarative part of the **LfP** main diagram consists of:

- *general model information.* Model name, author, version, associated UML model.
- *declaration of constants and new data types.* NBC, NBS and NBZ constants respectively define the number of conveyers, segments and crossing zones. A new type (station) defining valid station identifiers. Two enumerated types (start_bound and end_bound) representing valid bounds of a segment in terms of stations (stations are numbered). We assume here that there is only one station per segment. This may be changed without modifying the structure of the **LfP** model.
- *declaration of static class instances with their initial context.* We find two conveyer instances, four segment instances and three crossing zone instances.
- *a list of properties to be verified for the system.* These declarations belong to the property view of the system. We provide some interesting properties in Section 4.3.

4.2.2. Description of the SC Class

Figure 8 presents the SC class. The declarative part specifies for each connected media binder, the list of accepted messages (marked as in) and possible outgoing messages (marked with out). According to the sequence diagrams of the two scenarios (Figure 5 and 6), a SC class instance may receive an entrance demand (DEM) or a notification message (OUT). A segment controller may reply to the demanding conveyer controller using two alternative messages: entrance authorization (AUT) or entrance reject (REF). It also sends GO to let a waiting conveyer come in from a crossing zone.

```
messages:
 from SC_channel.server
  in DEM();
  in OUT();
  out AUT();
  out REF()
  out GO();
 from FIFO_channel.in
  out WRITE (in opaque data);
 from FIFO_channel.out
  in READ (out opaque data);
internal_methods:
procedures:
```

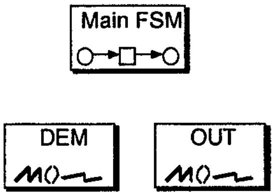


Figure 8 : The SC (Segment Control) class.

The main **LfP**-FSM (Figure 9) specifies the execution contract of the SC class. Its purpose is to merge together the two alternative behaviors corresponding to the execution scenarios from Figure 5 and Figure 6.

The main **LfP**-FSM states that DEM and OUT methods should be mutually exclusive. Moreover, OUT can be executed only if the segment contains a conveyer (segment state is full).

The main **LfP**-FSM declares three local variables (duplicated in any class instance): status represents the execution state of a class instance (empty or full); index stores the number of pending demands; HID represents the class instance identifier. HID corresponds to a unique instance identifier.

When constructing new class instances, all context variables have to be initialized.

```
context :
 local status is ('empty', 'full');
 local index := 0 is integer range 0..NBC;
 local HID is integer range 1..NBS;
```

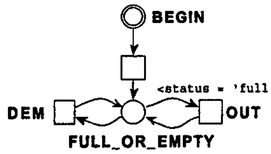


Figure 9 : The main LfP-FSM of the SC class.

Figure 10 presents the DEM method **LfP**-FSM. It defines conveyer_ID, a local variable used to store the identifier of a demanding conveyer.

The DEM execution contract has two branches:

- When the segment is empty, access is granted and the segment state changes to full.
- When the segment is full, the query is stored in a FIFO media for further process and the index of pending demands is incremented. Then, a negative response (REF) is sent back to the conveyer.

Communication with the FIFO has oneway asynchronous message passing semantics.

context : **local** conveyer_ID := 0 **is integer range** 0..NBC;

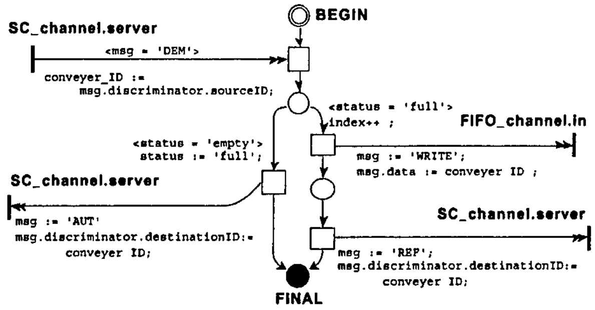


Figure 10 : LfP-FSM of the DEM method.

Figure 11 presents the **LfP**-FSM of the OUT method. As for DEM, it also contains a local variable to store conveyer identities. Behavior of the OUT method consist of two alternative branches:

- If there is no pending request (index = 0) then status of the segment is changed to empty.
- If there is at least one pending requests, the oldest demand is retrieved from the FIFO and a GO message is forwarded to the corresponding conveyer.

context : **local** conveyer_ID :=0 **is integer range** 0..NBC;

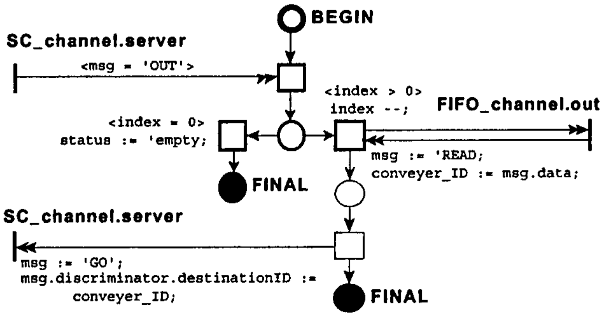


Figure 11 : LfP-FSM of the OUT method.

Due to space reasons the **LfP** representations of CC and CZC classes are not presented in this paper.

4.2.3. Description of the SC_channel media

The structure of the `SC_channel` media is presented below in Figure 12. Its context consists of two local variables: `client_ID` represents the identifier of the connected CC class and `message` is used to encapsulate the contents of an incoming message.

`SC_channel` has two binders (`client` and `server`) through which it is connected to a client (a CC class instance) and to all server instances together (all CS class instances). The connected conveyer is the client (`client` multiplicity is one) and all connected segments are servers (multiplicity of the `server` binder is marked as `all`).

The media may transport several messages. Possible messages and their parameters are enumerated for each binder. The communication is asynchronous through both binders.

When receiving a message through the client binder, the media dispatches it to the concerned server. This is achieved using a simple copy of the message contents from the incoming binder to the output one.

In order to match an incoming message from a server, the message destination should be identical to `client_ID`.

```
context :
 local client_ID is integer range 1..NBC;
 local message is opaque;

binders:
client: asynchronous ;            server: asynchronous;
       multiplicity := 1 ;               multiplicity := all ;
       messages: in  DEM ();             messages: out DEM ();
                 in  OUT ();                       out OUT ();
                 out AUT ();                       in  AUT ();
```

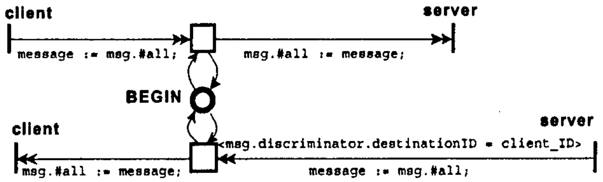


Figure 12 : LfP-FSM of the SC_channel media.

4.3. Properties to be Verified

Two kinds of properties may be considered with regards to modular specification:

- properties local to a module,
- properties global to the complete specification.

Local properties concern the internal behavior of a component independently from its environment. If we consider the `SegmentControl` class, local properties express the link between the demand of a conveyer and the answer of the segment, such as:

i. if SegmentControl gets request to enter an empty segment, it answers "OK" to the conveyer,
ii. if SegmentControl gets request to enter a full segment, it answers "REF" to the conveyer.

Global properties concern the behavior of the complete system; verification thus requires the specification of the entire system. An example of such a property is :

iii. the system is deadlock free.

5. FORMAL VERIFICATION

LfP specifications cannot be used «as is» to perform formal verification. Thus, a translation into a verification language is necessary. The generated formal specification is not as easy to read as the one in **LfP**, but handles automated formal verification.

It is usually impossible to perform formal verification without abstraction and reduction of the system at the formal level. However, as most abstractions and reductions rely on the semantics of the property to be verified, we produce one formal specification per property to verify. Thus, the obtained formal specification is equivalent to the **LfP** one regarding the considered property.

We choose well formed colored Petri net **[3]** because, in addition of excellent capabilities for the description of concurrent systems, they support both structural and behavioral verification methods.

Let us use the conveyers example to illustrate validation of the behavioral property stated in Section 4.3. To validate this system, we used CPN-AMI, a Petri net based CASE environment **[11]**.

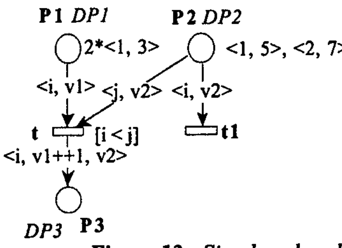
5.1. Colored Petri Nets

This section informally presents colored Petri nets.

A colored Petri net is a 5-uple <P, T, Pre, Post, Types, M_0> where:

- P is a set of places (depicted by circles).
- T is a set of transitions (depicted by rectangles).
- Pre[t] is the precondition function for transition t.
- Post[t] is the postcondition function for transition t.
- Types is the set of basic types. A basic type is a finite set.
- M_0 is the initial marking.

Figure 13 depicts a simple colored Petri net with 3 places (P1, P2 and P3) and 2 transitions (t and t1).



```
class
  Id is 1..3;
  Value1 is 1..10;
  Value2 is 4..15;
domain
  DP1 is <Id, Value1>;
  DP2 is <Id, Value2>;
  DP3 is <Id, Value1, Value2>;
var
  i, j in Id;
  v1 in Value1;
  v2 in Value2;
```

Figure 13 : Simple colored Petri net example.

To each place p, a domain Dom(p) is associated: Dom(p) is the cartesian product of some basic types. In Figure 13, basic classes are id, Value1 and Value2. The domain of P1 is the cartesian product of Id and Value1, the one of P2 is the cartesian product of Id and Value2 and the one of P3 is the cartesian product of Id, Value1 and Value2

A marking M(p) is associated to each place p: M(p) is a multi-set over Dom(p). Therefore, a marking M is the function that associates a marking to each place p of P. An element of a marking in a place is called a token. In Figure 13, the initial marking associates:

- two tokens having the <1, 3> profile to P1,
- tokens <1, 5> and <2, 7> to P2,
- the empty multi-set to P3.

Pre and Post functions describe how a marking is modified when an action is performed. Since actions are associated to transitions, instead of «an action is performed» we say: «a transition is fired».

To each transition, a set of variables Var(t) is associated. Each variable is defined over a basic type. In Figure 13, Var(t) = {i, j, v1, v2} and Var(t1) = {i, v2}. Variables i and j are defined over the basic class Id, variable v1 is defined over Value1 and variable v2 is defined over Value2

Let us call a binding of t the association of a value to each variable of Var(t). Let x a binding of t, Pre[t][p, x] returns a multi-set over Dom(p). A transition t can be fired for a marking M iff:

- constraints over the binding are satisfied (they are called guards),
- Pre[t][p, x] is included in M(p) for all p of P,

Post[t][p, x] also returns a multi-set over Dom(p). If t can be fired for binding x, then a new marking M' can be computed: M'(p) = M(p) - Pre[t][p, x] + Post[t][p, x]. Since a variable may appear in many post or preconditions, it is useful to define the successor (++n) and the predecessor (--n) functions.

In Figure 13, many bindings can be found for transition t, like i = 3, j = 5, v1 = 7, v2 = 6. However, t cannot be fired for this binding since <3, 7> is not a token in P1 for the initial marking. The following binding allows t to be fired : i = 1, j = 2, v1 = 3, v2 = 7 (token <1, 3> belongs to P1 and token <2, 7> belongs to P2, there is no precondition for P3 and the guard is satisfied since i < j). When t has been fired a new marking M_1 is computed:

- P1 contains the token <1, 3>,
- P2 contains the token <1, 5>,
- P3 contains the token <1, 4, 7>.

From this new marking no binding can be found for t to be fired (the only possible binding would be i = 1, j = 1, v1 = 3, v2 = 5 and it does not satisfy the guard i < j). Figure 14 shows the reachability graph of the net figure Figure 13. The double circled state corresponds to the initial marking (M_0) of the net.

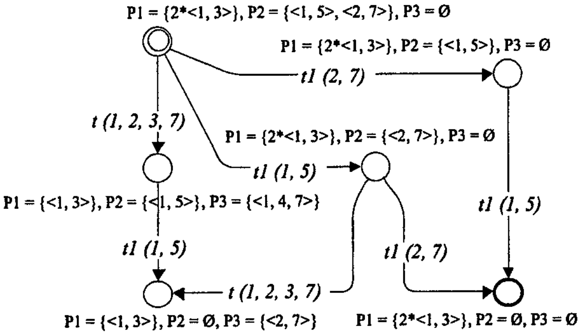

P1 = {2*<1, 3>}, P2 = {<1, 5>, <2, 7>}, P3 = Ø
P1 = {2*<1, 3>}, P2 = {<1, 5>}, P3 = Ø
t1 (2, 7)
t (1, 2, 3, 7)
P1 = {2*<1, 3>}, P2 = {<2, 7>}, P3 = Ø
t1 (1, 5)
P1 = {<1, 3>}, P2 = {<1, 5>}, P3 = {<1, 4, 7>}
t1 (1, 5)
t1 (1, 5)
t1 (2, 7)
t (1, 2, 3, 7)
P1 = {<1, 3>}, P2 = Ø, P3 = {<2, 7>}
P1 = {2*<1, 3>}, P2 = Ø, P3 = Ø

Figure 14 : Reachability graph of the simple colored Petri net.

5.2. From L*f*P to Colored Petri Nets

We have to ensure that results computed at the Petri net can be translated into **L*f*P** terms. Therefore, the translation process has to preserve the component structure of **L*f*P** models. This strategy also enables modular verification when it is possible (e.g. for local properties).

Therefore, we work at the module level (modules are deduced from **L*f*P** classes). We then compose them to produce a complete Petri net of the system.This procedure has two main steps:

- Generation of Petri net modules from **L*f*P**-FSMs.
- Composition of Petri net modules

To illustrate the translation procedure, we consider the specification of the SegmentControl.

5.2.1. Generation of Petri Net Modules

Structure of the Petri net. To obtain the structure of the Petri net, we consider **L*f*P**-FSM of the input model:

- In the Main-FSM, transitions corresponding to methods are replaced by the corresponding **L*f*P**-FSM.
- Places initiating methods are identified with the BEGIN place in the corresponding **L*f*P**-FSM
- Method output places are identified with FINAL places in the corresponding **L*f*P**-FSM.
- We preserve at the Petri net level, names of **L*f*P** places and transitions. Unnamed **L*f*P** places and transitions are given an arbitrary name in the Petri net. Naming is requested by some verification tools.

If we consider the SegmentControl class, we obtain the Petri net of Figure 15. Black places and transitions are the one of the Main-FSM. Transitions DEM, t1, t2, t3 and places P1, P2 describe the method DEM. Transitions OUT, t4, t5 and place P3 describe the behavior of method OUT. Transition DEM (respectively OUT) are shared by the Main-FSM and the method DEM (respectively OUT). Channel_SC_server and FIFO_channel_out correspond to media.

The verification we consider does not matter with the implementation of communication channels. We may thus abstract their specification with a single place. However it requires their implementation to respect the following property: *channels are deadlock and loss free.* This assertion has to be inserted as an implementation note used by code generators.

Color-domains, valuations and initial marking. Once the structure of the Petri net obtained, it is necessary to define variables management using color classes and domains, variables, valuations and guards. A color domain representing the information required to determine the state of the system is associated to places in the Petri net model.

We thus consider the variables identified in FSMs:

- Information depicted by variables declared in the main FSM is associated to place,
- Places derived from methods are enriched by local variables.
- Information in a channel contains three parts : the source, the destination and the value of the message.

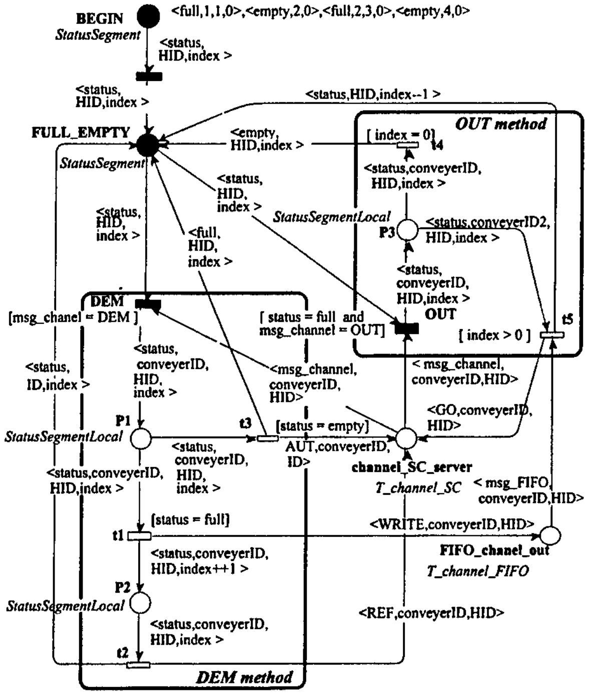


Figure 15 : Petri net of the SC (SegmentControl) class.

Let us illustrate this on the Petri net in Figure 15 and the corresponding declarative part in Figure 16. Places of the main FSM carry: status of a segment (empty or full), identity of the segment (integer between 1 and 4) and an index that indicates the number of conveyer waiting for segment release. This information is represented by the StatusSegment domain. Places derived from methods also contain the identity of the conveyer willing to enter the segment. Therefore the new domain, StatusSegmentLocal, is defined. Each of the two communication channels has its own domain (T_channel_SC and T_channel_FIFO).

The following declaration contains the classes, domains and variables that are necessary for the description of the SegmentControl class.

```
CLASS
  Tstatus is [empty,full];
  TCC is 1..2;
  TSC_CZC is 1..4;
  Tindex is 0..2;
  Tmsg_channel_SC is [OUT,DEM,AUT,REF,GO,VIDE,OK,PB];
  Tmsg_FIFO_channel is [ WRITE,READ ];
DOMAIN
  StatusSegment is <Tstatus,TSC_CZC,Tindex>;
  StatusSegmentLocal is <Tstatus,TCC,TSC_CZC,Tindex>;
  T_channel_SC is <Tmsg_channel_SC,TCC,TSC_CZC>;
  T_channel_FIFO is <Tmsg_FIFO_channel,TCC,TSC_CZC>;
VAR
 status in Tstatus;
 conveyerID2 in TCC;
 conveyerID in TCC;
 HID in TSC_CZC;
 index in Tindex;
 msg_channel in Tmsg_channel_SC;
 msg_FIFO in Tmsg_FIFO_channel;
```

Figure 16 : declarative part of the SC (SegmentControl) class.

Arcs valuation and transitions guards are also deduced from the **L/P** specification.Let us consider transition DEM in Figure 15.

To be fired it requires one token from place FULL_OR_EMPTY (domain StatusSegment) and one token from channel_SC_server (domain T_channel_SC). The arc from place FULL_OR_EMPTY to DEM is valuated by the tuple <status,HID,index>; the one from channel_SC_server to DEM is valuated by the tuple <msg_channel, conveyerID,HID>.

So, to fire DEM, a message must be sent to a segment identified as full or empty (this segment is not responding to a request) and HID variables must be the same in both valuation. The token stored in the DEM output place is a combination the input: <status,conveyerID,HID,index>.

Transition t3 authorizes (t2 refuses) the entrance in the segment; t3 has the following guard [status = empty] (respectively [status = full] for t2). These guards correspond to the preconditions defined in the **L/P**-FSM (Figure 10).

This way, places domain, arcs valuation and transitions guard are computed from the **L/P** specification. Petri net of Figure 15 and declaration in Figure 16 represent the complete Petri net specification of SegmentControl Class.

The initial marking of the Petri net corresponds to the static instantiation of classes. For SegmentControl, we indicate which are the full segments; for ConveyerControl we indicate the segment identifier where each conveyer is and, for we indicate that all CrossingZoneControl instances are empty.

Petri net reduction. As the Petri net is automatically synthesized from the **L/P** specification, its structure may be not optimized. Some reductions are possible regarding the class of properties to verify. These reductions concern the Petri net structure. Therefore, combinatorial explosion of the corresponding state graph is reduced and verification becomes easier.

If we consider deadlock freeness, reduction techniques presented in **[6]** can be applied. If we consider verification of temporal properties, reduction techniques presented in **[16]** are necessary.

The first technique is compatible with deadlock freeness property (property iii.), the second one is compatible with

Properties i. and ii. Some reductions, as the following one, belong to the two techniques. The rule we apply aims to identify two transitions (t_a and t_b) where the bindings of t_a depends only on the bindings of t_b. Such a reduction is possible between transitions DEM and (t1 and t3).

Figure 17 shows the reduced Petri net. Black transitions replace the three ones that have been reduced. The place between DEM, t1 and t3 has been suppressed.

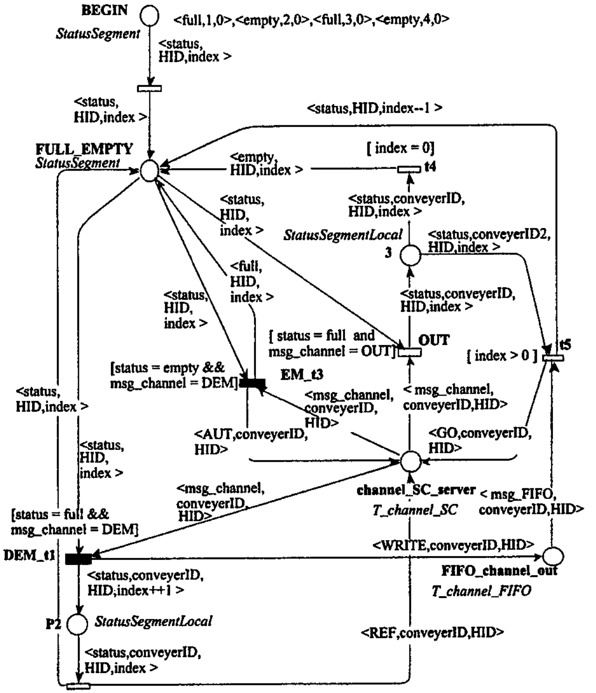


Figure 17 : Reduced Petri net of the SC (SegmentControl) class.

5.2.2. Composition of Petri Net Modules

Composition. The composition of modular Petri nets is obtained by identification of channel binding points. In our case, as each channel is represented by a single place, we perform the fusion of all the places representing the same channel.

Abstraction of system environment.To verify our system, we need a representation of its environment. At the verification level, this environment consists of the Command class (Figure 4) and channel_command.

Due to possible communications between Command and CC classes, this environment can be represented a generator of messages coming from the communication channel. Command sends all possible messages since we have no constraints on it. Therefore, even if we consider a particular initial configuration, the evolution of the system leads to all possible configurations.

The complete model.The assembled model, obtained by composition of reduced modules, contains 20 places, 28 transitions and 92 arcs.

5.3. Verification of properties

We consider two types of properties: local and global.

5.3.1. Local properties

Let us consider SegmentControl with properties i. and ii. The formal language we use to express such properties is a temporal logic [5], properties i. and ii. are interpreted as :

- ***i.:*** if transition DEM is fired with the binding (status = empty, conveyerID, HID, index, msg_channel = DEM) then place channel_SC_server will eventually contain the token <AUT,conveyerID,HID>,
- ***ii.:*** if transition DEM is fired with the binding (status = full, conveyerID, HID, index, msg_channel = DEM) then place channel_SC_server will eventually contain the token <REF,conveyerID,HID>

To verify such a property, it is not necessary to consider the entire system specification. The Petri net of ControlSegment class associated to an abstraction of its environment with an adapted initial marking is sufficient.

We use PROD **[20]**, a model checker dedicated to colored Petri nets and integrated as a component in CPN-AMI **[11]** to verify the temporal logic specification of these porperties.

These properties are verified. Once all classes individually verified, we can consider global properties.

5.3.2. Global Properties

We now want to verify property iii. Theorem provers are able to check such a property without considering a specific instantiation of the system (e.g. a given number of segments and conveyers). However, such proofs cannot be automated.

To enable automated proofs based on the reachability graph, we have to instantiate the system. Such instantiations can be deduced from the expected size of the system. A well accepted strategy is to start with a small number of resources and components to check if the property is correct. Then we use realistic dimensions of the system for a safer verification.

So, let us start with two conveyers and four segments. We first compute the reachability graph of the complete Petri net and look for terminal nodes (i.e. specification deadlocks). We also used PROD to evaluate this property.

The computed reachability graph holds 3072 nodes, 6209 arrows and 33 terminal nodes. Thus, our specification is not correct. PROD helped us to extract a path between the initial state and one of the terminal nodes. This path builds a scenario explaining why our specification is not deadlock free.The scenario is the following :

- Initially conveyer 1 is in segment 1 and conveyer 2 is in segment 3,
- Conveyer 2 asks for segment 2 and enters in it.
- Conveyer 1 then asks for segment 2 and is not allowed to enter it.
- Conveyer 1 therefore asks to enter in crossing zone 1.
- It is allowed to enter this crossing zone which is immediately set to occupied even if conveyer 1 has not yet left segment 1.
- Conveyer 2 then asks for segment 1, it is not allowed to enter since conveyer 1 has not yet left the place.
- Therefore it asks to enter crossing zone 1. This raises a problem and the system stops.

This deadlock is due to asynchronous communication between classes : conveyer 1 has not yet considered the authorization from the crossing zone since the crossing zone considers it is already in. Then, our specification does not ensure that the number of occupied segments and crossing zones remain equal to the number of conveyers. The verification process shows an implicit property that should be explicitly expressed in the **L/P** verification view.

To solve this problem, a communication protocol between classes has to ensure that when a crossing zone (or a segment) accepts a conveyer's demand, this conveyer is no more considered as being in the crossing zone (or segment) it is leaving. Transactions ensure such a property. THus, the systems designer should update the **L/P** specification according to this observation. Such an operation corresponds to what we called «formal debug» in Figure 1.

6. CONCLUSION

In this paper, we have presented an evolutionary prototyping methodology that promotes formal verification and debugging of a specification as well as code generation of distributed programs.

This methodology relies on **L/P**: a formalism offering structuration capabilities and having a precise semantics suitable for the description of interaction between components of an embedded distributed system. The strong semantical definition of **L/P** aims to eliminate problems observed on a standard notation such as UML when it comes to code generation and formal verification. However, **L/P** remains connected to UML since we consider it as an additional diagram. Some parts of this diagram may be deduced from classical UML diagrams but system designers have to provide new information regarding cooperation between classes in the system.

We illustrated our methodology on a small example: a conveyer system. This example showed that non-trivial errors can be detected on a system that appears to be correctly described. The detected problem deals with sophisticated behavioral aspects of the system which are due to some unspecified aspects on the system (here, some communication issues were not properly stated).

Based on the study presented in this paper, it appears that our methodology has some «nice» capabilities when designing a system:

- It is connected to a standard UML-based approach. In our methodology, UML design fits the early conception.
- **L/P** is used as a basis for detailed description of the system and a basis for code generation and formal verification.
- Our transformation techniques preserve a strong correspondence between the model level (**L/P**), the formal level and the program level.
- The use of formal methods to check properties may be hidden to the end user. Then, it can be used by engineers having a low knowledge on formal methods.
- Formal verification techniques enable formal debug at the model level. Then, if we assume that no bug is introduced by code generators, system implementation should behave according to the verified properties.

Our methodology is partially implemented in CPN-AMI, a Petri net based CASE environment.

7. REFERENCES

[1] J.R. Abrial, "The B-book", Cambridge University Press, 1995

[2] M. Björkander, "Graphical Programming Using UML and SDL", in Computing Practice, Vol. 33, No. 12, December 2000, <http://www.computer.org/computer/co2000/rztoc.htm#rz030>

[3] G. Chiola, C. Dutheillet, G. Franceschini & S. Haddad, "On Well-Formed Coloured Nets and their Symbolic Reachability Graph", High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991

[4] M Elkoutbi, R. Keller :"Modeling Interactive Systems with Hierarchical colored Petri nets", In Proceedings of the 1998 Advanced Simulation Technologies Conference, pages 432-437, Boston, MA, April 1998. The Society for Computer Simulation International. HPC98 Special session on Petri-Nets.

[5] E. Emerson, "Temporal and Modal Logic" Handbook of Theoretical Computer Science, Chapitre 16, pages 995-1072 , Elsiever Science, 1990

[6] S. Haddad, "A Reduction Theory for Coloured Nets", LNCS : High Level Petri Nets. Theory and Application. Edited by K. Jensen , G. Rozenberg, Springer Verlag 1991

[7] V. Issarny, T. Saridakis and A. Zarra : "Multi-view Description of Software Architectures", in Proceedings of the 3rd ACM SIGSOFT International Software Architecture Workshop, pages 8184, November 1998, http: //www.irisa.fr/solidor/doc/../doc/ps98/isaw98.ps.gz

[8] ITU-T : "Open Distributed Processing", X.901, X.902, X.903 and X.904 standards, <http://www.itu.int/itudoc/itu-t/rec/x/x500up>

[9] N. Leveson, "Software Engineering: Stretching the Limits of Complexity", Communications of the ACM, Vol 40(2), pp 129-131, February 1997.

[10] J. Lilius, I. Porres Paltor :" vUML: a Tool for Verifying UML Models", in Proceedings of the 14th IEEE International Conference on Automated Software Engineering, October, 1999. <http://dlib.computer.org/conferen/ase/0415/pdf/04150255.pdf>

[11] the LIP6-SRC team, the Mars project homepage, <http://www-src.lip6.fr/logiciels/mars/>

[12] Luqi & J. Goguen : "Formal Methods: Promises and Problems", IEEE Software, Vol 14, N°1, pp 75-85, January 1997.

[13] Luqi, V. Berzins, M. Shing, R. Riehle and J. Nogueira : "Evolutionary Computer Aided Prototyping System (CAPS)", in Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34), August 2000, Santa Barbara, California

[14] N. Medvidovic, Richard N. Taylor : "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, vol. 26, no. 1, january 2000. <http://dlib.computer.org/ts/books/ts2000/pdf/e0070.pdf>

[15] OMG, "OMG Unified Modeling Language Specifiacation", version 1.3, June 1999, <http://www.omg.org/cgi-bin/doc?ad/99-06-09.zip>

[16] D. Poitrenaud and J.-F. Pradat-Peyre, "Pre and Post-agglomerations for LTL Model Checking", proceedins of 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, June 2000, pages 387-408, Springer-Verlag

[17] D. Quartel, M. van Sinderen, L. Ferreira Pires : "A model-based approach to service creation", in Proceedings of the Seventh IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, pages 102-110. IEEE Computer Society, 1999. <http://wwwhome.cs.utwente.nl/~quartel/publications/Ftdcs99.pdf>

[18] D. Regep, F. Kordon, "**L/P**: a specification language for Rapid prototyping of Concurrent Systems", to appear in proceedings of the 12th IEEE International Workshop on Rapid System Prototyping, Monterey, June 2001

[19] J. Saldhana and S. M. Shatz, "UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis," Proceedings of the Int. Conference on Software Engineering and Knowledge Engineering (SEKE), Chicago, July 2000, pp. 103-110.

[20] K. Varpaaniemi, J. Halme, K.Hiekkanen & T.Pyssysalo, "PROD reference manual", Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995

A Model Checking Framework for Layered Command and Control Software

Kathi Fisler
Department of Computer Science
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
sk@cs.brown.edu

Don Batory
Department of Computer Science
University of Texas at Austin
dsb@cs.utexas.edu

Jia Liu
Department of Computer Science
University of Texas at Austin
jliu@cs.utexas.edu

March 21, 2001

Abstract

Most existing modular model checking techniques betray their hardware roots: they assume that modules compose in parallel. In contrast, layered software systems, which have proven successful in many domains, are really quasi-sequential compositions of parallel compositions. Most such systems demand and inspire new modular verification techniques. This paper presents algorithms that exploit a layered (or feature-based) decomposition to drive verification. Our technique can verify most properties locally within a layer; we also characterize when a global state space construction is unavoidable. This work is motivated by our efforts to verify a military fire simulation and support software system called FSATS.

1 Introduction

Today's software applications are modularized around objects that collaboratively interact to provide the functionality of an application. This is the fundamental starting-point for contemporary object-oriented design as well as contemporary modular model checking techniques [15, 19, 23, 27].

An alternate form of modularity centers around *features* rather than objects. Programmers design and construct applications by introducing one feature at a time. Each feature adds new capabilities and responsibilities to previously existing objects and introduces new objects to a design. A characteristic of features is that they are largely independent: this substantially reduces application complexity (because the concerns and implementation details of one feature are separable from those of others) and increases application extensibility (because new features can be easily added and unwanted features removed). Feature-oriented design is a form of step-wise refinement in which the refinements are entire features, rather than low-level changes to individual statements.

Many research efforts now approach design through features, including layers [5], collaborations [26], aspects [22] and units [14]. In this paper, we call them *layers* to evoke the visual imagery of feature-based refinement. Layers have been particularly successful in software product-lines, where each application of a product-line is defined by a unique combination of features. A brief sampling of successful designs in this vein includes a military command-and-control scenario simulator [4], a programming environment [13], network protocols and database systems [5, 6, 33], and verification tools [16, 30].

The success of layered designs at *implementing* software product-lines suggests a tantalizing prospect: they may also assist in *validating* product-lines. Layers have well-defined interfaces that permit their composition to build larger systems. Layers tend, at least in principle, to obey the characteristics of components [17, 20, 31], such as separate compilation, multiple instantiability and external linkage. Perhaps we can verify each layer individually, and perform cross-layer verification when composing layers.

As a case study, we are especially interested in a layered system called FSATS, which implements a military command-and-control scenario simulator [4]. In particular, we wish to apply model checking to FSATS. FSATS is a particularly good candidate for model checking for two reasons. First, the design includes specifications of state machines [4], which eliminates the problem of deriving such state machines from the software. Second, the system has several temporal properties (for instance, *every accepted mission eventually results in a weapon firing*) that are especially amenable to model checking.

FSATS is a complex collection of over 19 layers that can be composed independently to form scenario simulators. This immediately makes the straightforward application of model checking impossible, due the combinatorial number of possible systems, and the sizes of the larger compositions. FSATS has thus inspired our research into new forms of modular algorithmic verification. As FSATS is too complex to serve as a running example in this paper, we discuss it briefly to elicit its key characteristics, then illustrate our development on two simple examples that distill these characteristics.

The rest of this paper is organized as follows. Section 2 discusses prior work on modular verification and its relationship to our work. Section 3 discusses FSATS and presents our methodology. Section 4 presents conclusions and discusses avenues for future work.

2 Background and Related Work

Model checking is a technique for proving logical properties of systems [9]. Its successful application to hardware makes its use on software systems an attractive proposition. In a canonical model checker, a design is represented as a (finite) state machine, while properties are usually expressed in variants of temporal logic. Model checkers handle designs consisting of several machines running in parallel by automatically computing the cross-product of the machines, then applying their algorithms to the resulting single machine; we exploit this feature in section 3. For an extensive survey of model checking, we refer the reader to the book by Clarke, Grumberg and Peled [9]. In the rest of this paper, we assume a basic familiarity with model checking.

Model checking algorithms vary with the logic of properties. Our work extracts properties of layers by examining the labels on interface states. This assumes the model checker uses state labeling, which is the technique employed for branching-time temporal logics such as CTL. To simplify the development, we present our algorithms assuming an explicit representation of the state space of a system. In practice, many model checkers represent state symbolically rather than explicitly [25]. Our algorithms are insensitive to this difference; indeed, we performed the verification tasks in this paper on a model checker employing symbolic representations [32].

Several researchers have described techniques for modular verification of designs [15, 19, 23, 27]. These techniques are based on a hardware-oriented notion of modularity, in which modules are composed in *parallel.* For instance, one module might be a CPU, while another module represents a floating-point co-processor. The research then shows how to ensure the preservation of individual properties about the CPU or floating-point processor; using these techniques to prove properties involving both devices requires substantial experience, and is not always possible. These results do not apply to most software systems, where control flows sequentially between modules.

Some preliminary research [2, 10, 24] has begun to consider modular verification with sequential, rather than parallel, control flow. The original work [24] handles systems with only one state machine; it also lacks a design framework, such as layered design, that drives the decomposition of the system. Subsequent work uses hierarchical state machines [2] and StateCharts [10] to provide this decomposition, but the resulting systems are still monolithic. In contrast, we analyze systems with three key distinguishing features:

- Each layer introduces a feature that was not previously present in the system; the layer does not simply refine existing features.
- Layers are developed without knowledge about all the other layers that may exist in a final, composed system.
- Each layers (unit of sequential composition) encapsulates simultaneous extensions to multiple state machines.

The work by these other authors does not even admit these design possibilities. Alur and Yannakakis cite the problem of sequential verification over multiple state machines as open for future work [2]. Furthermore, they do not discuss how to handle systems that involve quasi-sequential composition of parallel compositions, such as exist in FSATS. Alur *et al.* discuss analysis techniques for sequential refinements within modules that are composed in parallel (this

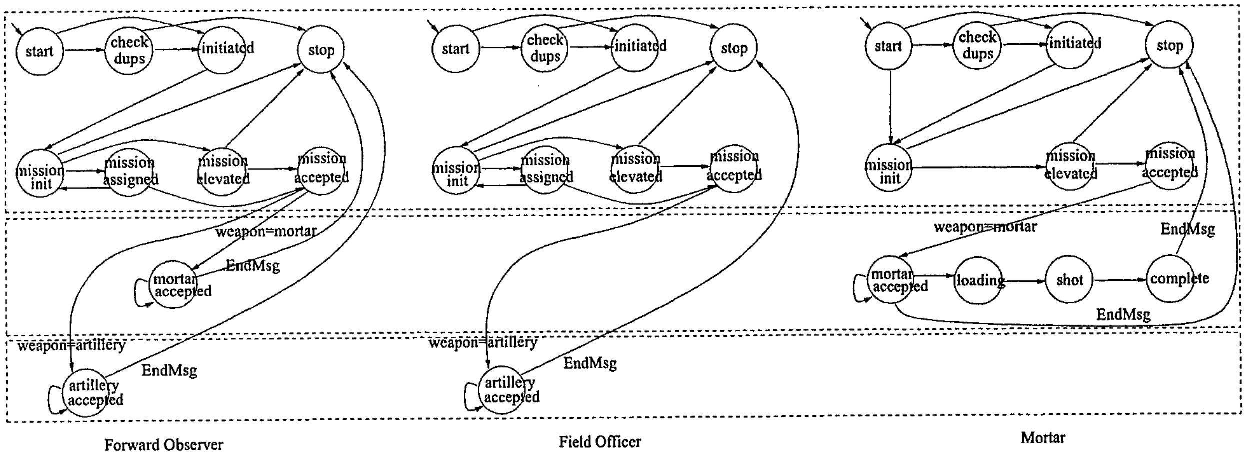


Figure 1: A sample of layered state machines from FSATS. The top layer is the base system, the second adds missions that fire mortars, and the third adds missions that fire artillery. The full design contains state machine hierarchies for the artillery as well as for other personnel in the command hierarchy.

work uses the term "behavioral hierarchy" for refinements within modules and "architectural hierarchy" for parallel compositions of modules) [1]. The critical difference between their work and ours is that theirs does not support *coordination* between sequential refinements across modules. Our work, in contrast, considers verification for layers that gather related sequential refinements into modules. Encapsulating related refinements in layers allows us to verify properties of entire features in isolation from other features, even when those features cross-cut several actors (*i.e.*, objects). Without a layered architecture, isolating this information from across parallel modules is difficult if not impossible.

3 Verifying Layered Software Systems

3.1 FSATS: An Example of Layered Design

FSATS is a command and control simulator. At core, it consists of a series of protocols for selecting weapons to fire at potential targets. The actors in FSATS are various military personnel (forward observers, field officers, brigade commanders, etc) arranged in a command hierarchy and the weapons at their disposal. Observers repeatedly identify potential targets and send messages along the command hierarchy to initiate missions against the targets. The personnel in the hierarchy accept or forward missions depending upon the weapons at their disposal. In the FSATS implementation, each potential target spawns a new thread in which to execute the protocol for handling that target.

Batory *et al.* have presented a layered design of FSATS, written largely as a set of layered state machines [4]. Figure 1 shows a sample of the layered state machines that comprise FSATS. A careful look at the machines and the code shows several characteristics that are potentially interesting from a model checking perspective:

- Each layer's (extension) state machines compose sequentially with their corresponding base machines.
- Each layer (extension) attaches to a common start and end point from the base layer.
- The conditions under which control enters a particular layer are similar across all actors and are closely coordinated through message passing.
- The state machines essentially synchronize at the end of a mission (on the EndMsg messages) right before the involved actor threads terminate.

These four observations drive the methodology and algorithms presented in this paper. The combination of these characteristics enable a powerful, modular approach to verification in which we verify layers individually and reason about property preservation under layer composition.

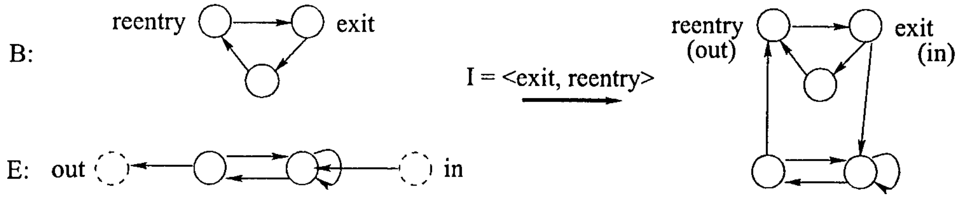


Figure 2: Composition of a base system B with an extension E via an interface.

3.2 A Model of Layered Design

We view a design as a set of classes, roughly one per actor in the system. A layer consists of a set of class extensions (*mixins* [7, 18, 28, 29, 34]) for the actor classes. The set of mixins in a layer relate to a common task, or *feature*, in the overall system (in FSATS, the features generally represent missions that utilize different weapons). This definition permits actor classes and mixins of arbitrary complexity. To make the problem of verification more tractable, we assume each actor class can be described as a state machine, and that each mixin extends an existing (base) state machine by adding nodes, edges, and/or paths between states in the base machine. State machine models of software arise from one of two sources: either the software is written in terms of state machines, as is true for many embedded software applications, or abstraction techniques derive state machines from the source code [11, 12]. FSATS is of the former flavor. Our work could adapt to the latter if the abstractions produce machines for which we could define meaningful interfaces between layers; accordingly, we regard the work on state machine abstractions as orthogonal to this paper.

Each base or composed system specifies interfaces, in terms of states, at which clients may attach extensions. We define interfaces formally below. In our experience, new features generally attach to the base system at common or predictable points, as Figure 1 illustrates; the set of interfaces is therefore small. This is important, as the interface states will indicate information that we must gather about a system in order to perform compositional verification of layers; a large number of interfaces might require too much overhead in our methodology.

Figures 3 and 6 show examples of base systems, layers, extensions, and interfaces; Sections 3.4 and 3.5 explain the examples in detail. The following formal definition makes our model of layered designs precise. The definitions match the intuition in the figures, so a casual reader may wish to skip the formal definition.

Definition 1 A *state machine* is a tuple $\langle S,\Sigma,\Delta,s_0,R,L\rangle$, where S is a set of states, Σ is the input alphabet, Δ is the output alphabet, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \times S$ is the transition relation (where $PL(\Sigma)$ denotes the set of propositional logic expressions over Σ), and $L : S \rightarrow 2^{\Delta}$ indicates which output symbols are true in each state.

Definition 2 A *base system* consists of a tuple $\langle M_1,\ldots,M_k\rangle$ of state machines and a set of *interfaces*. We denote the elements of machine M_i as $\langle S_{Mi},\Sigma_{Mi},\Delta_{Mi},s_{0_{Mi}},R_{Mi},L_{Mi}\rangle$. An interface contains a sequence of pairs of states $\langle\langle exit_1, reentry_1\rangle,\ldots,\langle exit_k, reentry_k\rangle\rangle$ Each $exit_i$ and $reentry_i$ is a state in machine M_i. State $exit_i$ is a state from which control can enter an extension machine, and $reentry_i$ is a state from which control returns to the base system. Interfaces also contain a set of properties and other information which are derived from the base system during verification; we describe these properties in detail in later sections.

Definition 3 An *extension* is a tuple $\langle E_1,\ldots,E_n\rangle$ of state machines. Each E_i must induce a connected graph, must have a single initial state with in-degree zero, and must have a single state with out-degree zero. For each E_i, we refer to the initial state as in_i and the state with out-degree zero as out_i. States in_i and out_i serve as placeholders for the states to which the layer will connect when composed with a base system. Neither of these states is in the domain of the labeling function L_i.

Given a base system B, one of its interfaces I, and an extension E, we can form a new system by connecting the machines in E to those in B through the states in I, as shown in Figure 2. For purposes of this paper, we assume that B and E contain the same number of state machines. This restriction is easily relaxed; the relaxed form allows actors to not participate in each new feature, or to allow new actors as required by new features. The mortar mission in FSATS (Figure 1, first extension layer), for example, does not augment the protocol of field officers. We also assume that the states in the constituent machines of base systems and extensions are distinct.

Definition 4 The composition of base system $B = \langle M_1,\ldots,M_k\rangle$ and extension $E = \langle E_1,\ldots,E_k\rangle$ via an interface $I=\langle\langle exit_1, reentry_1\rangle,\ldots,\langle exit_k, reentry_k\rangle\rangle$ is a tuple $\langle C_1,\ldots,C_k\rangle$ of state machines. Each $C_i = \langle S_{Ci},\Sigma_{Ci},\Delta_{Ci},s_{0_{Ci}},R_{Ci},L_{Ci}\rangle$ is defined from $M_i = \langle S_{Mi},\Sigma_{Mi},\Delta_{Mi},s_{0_{Mi}},R_{Mi},L_{Mi}\rangle$ and $E_i = \langle S_{Ei},\Sigma_{Ei},\Delta_{Ei},s_{0_{Ei}},R_{Ei},L_{Ei}\rangle$ as follows: $S_{Ci} = S_{Mi} \cup S_{Ei} - \{in_i, out_i\}$; $s_{0_{Ci}} = s_{0_{Mi}}$; R_{Ci} is formed by replacing all references to in_i and out_i in R_{Ei} with $exit_i$ and $reentry_i$, respectively, and unioning it with R_{Mi}. All other components are the union of the corresponding pieces from M_i and E_i. We will refer to the cross-product of $C_1,\ldots,C_k$ as the *global composed state machine.* Composed systems may serve as subsequent base systems by creating additional interfaces as necessary.

3.3 Verification Methodology

Our methodology is designed to support compositional verification of layered designs. Specifically, our methodology supports the following activities:

1. Proving a CTL property of an individual layer or composition of layers. This is easily done in the base system with existing techniques, but becomes more complicated in extension layers.

2. Deriving a set of constraints on the exit and reentry states of a layer that are sufficient to preserve a particular property after composition (the *preservation constraints*).

3. Proving that a layer satisfies the preservation constraints of another layer (or existing system). This activity is only meaningful if the preservation constraints were generated for the exit and reentry states to which the new layer will attach. We establish preservation by analyzing only the extension, not the composition of the extension and the existing system.

These activities correspond to a kind of modular verification, where the layers are modules. As in standard approaches to modular verification, we are interested in proving properties of modules and in preserving those properties upon composition with other modules.

We illustrate our methodology using two examples: a sportswatch and a communication protocol. The sportswatch design consists of a single actor; each collaboration therefore contains and extends only one state machine. This example motivates our interfaces and high-level approach to sequential layer composition. The communication protocol captures the key characteristics of FSATS identified in Section 3.1 and shows how our methodology extends to designs with multiple state machines in each collaboration. We have performed all verification runs cited in these sections using the described methodology and the VIS model checker [32]. Section 3.6 discusses pragmatic issues behind these runs.

3.4 Single-Machine Designs

Figure 3 shows a layered design of a sportswatch with timer and alarm features. The base system contains four display nodes: clock display, alarm time display, date display, and an alarm status display that supports toggling the alarm status. The first extension adds a timer which the user can reset, resume, and stop. The timer layer also supports a split timer for capturing time instantaneously. The second extension supports setting the alarm time; we omit layers for setting the clock time due to space constraints. Although both extensions add core functions, rather than optional features, we implement them as layers to allow a designer to include any of several possible implementations of these features in a final watch (as in a product-line architecture). The watch is controlled through two buttons (B1 and B2) and a mode switch that can be in the forward (ms-f) or back (ms-b) positions.

The base system should satisfy the property that one can always get to the display-clock state (written as AG EF *display_clock* in CTL). This property is easy to verify using a model checker. The base layer publishes one interface: $\langle dispclock, dispclock\rangle$, meaning that all extensions will start from and return to the *dispclock* state. Once we extend the base system with the timer, we must prove that adding the timer will not cause the display-clock property — which has already been proven of the base layer — to fail. We could compose the base system and timer layers and re-verify the property on the composed system. This approach, however, wastes the work that we have already done proving the property of the base layer; worse still, on a larger example, the composed design could be too large to model check. We therefore want to verify that the timer layer will preserve the property already proven of the base system without using the entire base system.

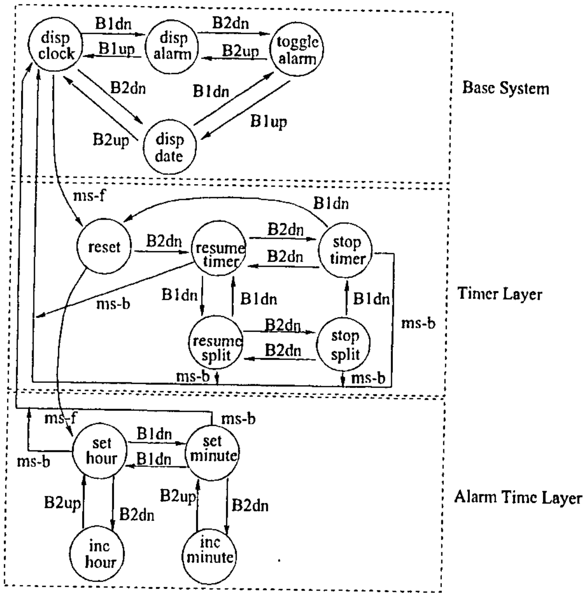


Figure 3: A collaborative design for a sportswatch.

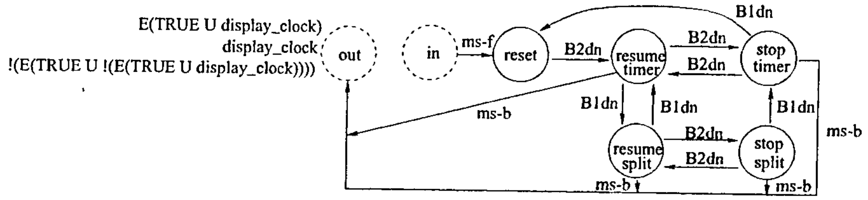


Figure 4: The timer extension with marking assumptions on the *out* state.

The classic CTL model checking algorithm [8] checks a property by marking each state with the subformulas of the property that are true in that state. After marking is complete, the formula is true of the design if its initial state is marked with the full property formula. If we can prove that an extension does not alter the markings of the base system states for a given property, then that property will hold in the composition of the base system with the extension as well. It suffices to show that the markings of the exit states in the base system interfaces are not altered, as all states which reach layer states do so through the exit state.

Given the base system interface (⟨*dispclock*,*dispclock*⟩ in this case) and a property to preserve (AG EF *display_clock*), we use a model checker to extract the set of subformulas of the property that mark each state in the interface; these markings can be stored with the interface, and need not be re-computed on each extension. The following three formulas mark *dispclock*:

- E*(TRUE* U *display_clock)*
- *display_clock* (this implies the previous formula)
- *!(*E*(TRUE* U *!(*E*(TRUE* U *display_clock))))* (equivalent to AG*(*EF*display_clock))*).

We must prove that the extension will preserve the markings on the exit state from the base system. The CTL model checking algorithm marks states based on the markings of its successor states. As some extension states have transitions to the reentry state (in the base system), we need the reentry state's markings to compute the markings on the extension states. Our verification algorithm consists of assuming that the *out* state of the extension has the same markings as the reentry state, deriving the markings on the *in* state, and checking that those markings are the same as on the original reentry state; this approach is consistent with the standard backwards-reachability approach to model checking. We derive the markings on the *in* state by checking a property of the form AG*(in* $\rightarrow$ ϕ*)* for each subformula ϕ of the property to be preserved. Figure 4 shows the sportswatch timer layer with the marking assumptions on *out*.

Model checking confirms that *in* retains the original markings of *dispclock*, so the property will be preserved upon composition.

In addition to the display-clock property, we can also verify that the timer layer (without the base layer attached) satisfies the property "once started, the timer can always be stopped" ($\mathsf{AG}(\mathit{start\text{-}timer} \rightarrow \mathsf{EF}\ \mathit{stop\text{-}timer})$). We view the timer layer as the base system and the base as the extension to verify that the base layer would preserve this property upon composition.

We also construct a composed system from the base layer and the timer extension, with interface $\langle \mathit{dispclock}, \mathit{reset} \rangle$. The interface states change after composition because the watch requires switching between modes to be deterministic; satisfying this constraint requires new layers to be entered from the timer layer, rather than the original base system. For both states in the interface, we record the markings necessary to satisfy the two properties already proven of the system. These markings arise from both verifying the properties of each layer and from verifying the preservation of the other layer's properties. For *dispclock*, the new set of interface markings is:

- $!(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ !(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{display\_clock}))))$;
- $\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{display\_clock})$;
- $\mathit{display\_clock}$;
- $!(\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ !((\mathit{starttimer} \rightarrow \mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{stoptimer})))))$;
- $(\mathit{starttimer} \rightarrow \mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{stoptimer}))$;
- $\mathsf{E}(\mathit{TRUE}\ \mathsf{U}\ \mathit{stoptimer})$

Using these markings, we verify that adding the alarm layer preserves the existing properties (displaying the clock and stopping the timer).

3.4.1 Summary of Algorithm on Single State Machines

In summary, the verification algorithm for the single state machine case is as follows:

1. Write the model for the extension, including the placeholder states *in* and *out*.
2. Assign the subformulas that marked the actual reentry state in the base system as labels of the placeholder reentry state (*out*).
3. Model check all of the subformulas of the original property in the placeholder reentry state (*in*). If *in* has exactly the same markings (restricted to subformulas of the property) as it did before the extension, the property will hold in the composed system.

This algorithm, whose correctness proof we defer to a forthcoming technical report, was independently derived by Laster and Grumberg for reasoning about sequential decomposition of finite state machines [24]. Its correctness depends in part on all reachable states in the composed design lying in either the base system or the extension (an obvious point in the single-machine case, but one which becomes interesting in the multiple-machine case). For checking preservation of purely existential properties, this algorithm is unnecessary because sequential composition trivially preserves such properties (a simple observation, but one which Laster and Grumberg [24] did not note).

For our experiments, we simulated this algorithm using the VIS model checker. VIS does not support this algorithm directly, as there is no way to seed *out* with the assumed marking. Instead, we were forced to include a transition from *out* to the entire base system model; we did not include transitions from the base system to *in*. We verified that the markings on the actual reentry state (*dispclock*) did not change under this operation. As *in* was not attached to the base system, this approach is sufficient to argue that the verification would have gone through with the seeded markings (and no base system) had VIS supported that operation. Section 3.6 summarizes the issues that arose trying to use conventional model checkers for this sort of modular verification.

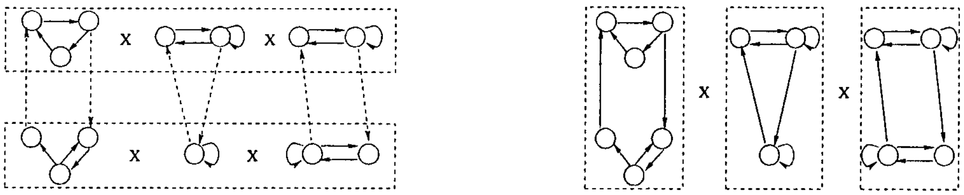


Figure 5: Two approaches to constructing composed systems.

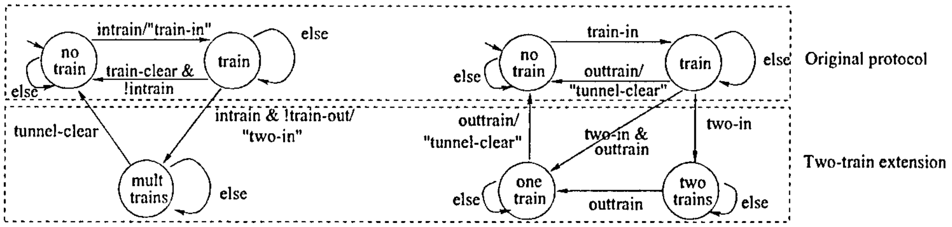


Figure 6: A collaborative design for a track-operator communication protocol.

3.5 Multiple-Machine Designs

The algorithm in Section 3.4.1, as well as prior research into verification under sequential composition, does not apply to FSATS because FSATS has multiple state machines in each layer. In practice, almost all interesting *collaborative* designs, by their very nature, will employ multiple state machines extensions per layer. When each layer contains a single state machine, extending a system with a layer corresponds to sequential composition of state machines. When layers contain multiple state machines, extending a system with a layer corresponds to a hybrid of sequential and parallel composition: the machines within a layer are composed in parallel (because they run together to implement a particular feature), but the layers themselves are composed in a quasi-sequential manner. The actual composition is not strictly sequential: this detail is at the crux of the verification problem for systems like FSATS.

Constructing a design by sequential composition is appealing because, as Section 3.4 shows, it supports independent verification of layers. Figure 5(left) shows a layered system constructed in this fashion. The construction provided in the formal model (the global composed state machine, Definition 4), however, is different. As Figure 5(right) illustrates, the construction first extends each base machine with its corresponding mixin, then composes the resulting machines in parallel. Clearly, we would prefer to compose systems according to the first construction because it supports layered verification. In order to do this, however, the first construction must produce the same global composed state machine (upto reachability of states) as the second! This relationship captures the crucial challenge in layered verification of designs with multiple state machines per layer. We must construct the parallel compositions representing each layer in such a way that composing them sequentially yields the state machine arising from Definition 4. This is possible only because most cross-product states in the composite system arise from cross-products within layers; this section notes the exceptions and how our methodology handles them.

This section motivates our algorithm for constructing parallel, compositions within layers. Our algorithm is designed to create parallel compositions that can in turn be composed sequentially with other layers. We describe the algorithm by illustrating its behavior on a small example. We also evaluate this algorithm's ability to verify properties of layers in isolation. While many layers (including the FSATS layers) can be verified in isolation under this construction, our motivating example illustrates a case where independent verification may fail. A property for which verification may fail must be verified in the composed system, rather than compositionally through the layers. We provide a characterization of these cases and a model-checking-based algorithm to determine whether properties can be verified compositionally. Section 3.5.1 presents our new example, which captures the salient characteristics of FSATS without necessitating as much explanation of the domain.

3.5.1 The Clayton Tunnel Protocol

We consider a layered design of a communications protocol between operators at either end of a train tunnel (see Figure 6). This protocol, taken from Holzmann's book [21], should already be familiar to those versed in the model

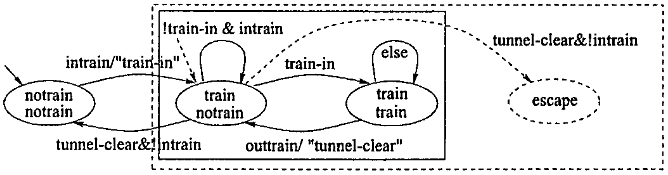


Figure 7: The cross-product state machine for the tunnel base layer. The exit subgraph for an interface containing both *train* states as exit states is enclosed in the solid box. The dashed box encloses the exit subgraph extended with an *escape* state for capturing the conditions under which control would leave the exit subgraph. The portion in the dashed box is part of the interface of the base system with both *train* states as exit states.

checking literature. Our design is derived from an actual communication protocol that was in use (and contributed to an accident!) in England in 1861. The two state machines model the human operators on either end of long train tunnel covering a one-way track. Unable to see one another, the operators communicate messages about the status of the tunnel. In the base layer, the operators communicate when trains are entering and exiting the tunnel. The inbound operator sends a *train-in* message to the outbound operator when a train enters the tunnel. The outbound operator sends a *train-clear* message to the inbound operator when a train exits the tunnel. The base layer consists of the protocol for exchanging these two messages.

The full protocol was designed to prevent two trains from ever being in the tunnel simultaneously (we omit the specific details from the model is this paper because they are irrelevant for our purposes). The accident that occurred arose because a second train entered the tunnel (in the same direction as the first train) before the first one left; although the inbound operator suspected the problem, the communication protocol was too weak to convey the situation to the outbound operator. One solution is to add messages to the protocol that convey this information accurately. The extension adds a *two-in* message from the inbound to the outbound operator; it also adds states to both operator machines so that the outbound operator does not send the *train-clear* message until both trains have left the tunnel.

Verifying this protocol requires a model of the trains that can enter and exit the tunnel. A model of the events that drive a protocol, but are not part of its definition, is called an *environment model.* The environment model for the tunnel protocol must generate reasonable train data; for example, no train should ever leave the tunnel before it enters the tunnel. For simplicity, we use an environment model containing two trains. Their only constraints are that the first train enters the tunnel before the second, and that both trains enter the tunnel before they exit the tunnel. This model is reasonable because the original protocol was such that at most two trains could be in the tunnel at once if the train drivers obeyed the rules of using the tunnel. We implement environment models as state machines. For the tunnel protocol, the model generates signals *intrain* and *outtrain* to indicate trains entering and leaving the tunnel.

Depending upon when trains enter and leave the tunnel, the operators may be inconsistent on their views as to whether there is a train in the tunnel. Given the base layer, we would like to prove that the inbound operator never livelocks thinking that there is a train in the tunnel (AG*(*EF*(inbb.state=notrain))*; this property requires all trains in the tunnel to eventually exit the tunnel, which we handle with a fairness constraint [9]. We can easily discharge this property of the base system; the challenge is to verify that the extension preserves it. For the extension, we wish to prove that once the inbound operator warns that there are two trains in the tunnel, it does not exit the extension until it receives a tunnel-clear message *(*AG *((inbmsg=two-in)* → A*(!(instate=out)* U *(outbmsg=tunnel-clear))))*.

3.5.2 Composing the Extension in Parallel

The extension consists of the two state machines in the dashed box in Figure 6 (though with *in* and *out* states, as in Figure 4). We could form a naïve parallel composition of these two machines using a standard cross-product procedure [9]. This construction assumes that both machines start in their initial states (the *in* states) simultaneously. This assumption, however, is not necessarily valid. For example, the inbound operator may notice the second train before the outbound operator has registered that there is a train in the tunnel (this synchronization problem arises in FSATS). Our parallel composition therefore needs additional information about the synchronization of the *in* states in the extension in order to construct a valid composition. Fortunately, we can derive this information from the base system. Given a set of exit states that form an interface in the base system, we can compute the subgraph of the base system that involves only the exit states; we then use this subgraph (which we call the *exit subgraph,* defined formally below) to drive transitions between the *in* states in the parallel composition. Figure 7 shows the exit subgraph for the tunnel protocol. To verify preservation of properties under sequential composition, the exit subgraph includes a state that indicates when a transition would have left the exit subgraph; this state is labeled *escape* in Figure 7. While in practice this subgraph could be large, these graphs are small in FSATS (and presumably similar systems) because the actors decide to enter a particular extension at roughly the same time based on a tight sequence of message passing. Section 3.5.3 discusses a similar problem on the reentry states.

The following steps construct the exit subgraph:

1. Construct the cross-product of the base system machines.

2. Restrict the cross-product states and transition relation to those states that contain at least one exit state from some state machine in the cross product.

3. Add a new state *escape* to the resulting graph. From every state in the exit subgraph, add a transition to *escape* enabled on each condition that causes a transition outside of the exit subgraph. There are no transitions out of *escape*.

4. Identify all states (other than *escape*) with no incoming edges as initial states of the exit subgraph.

In the general case, the exit subgraph might not be connected. In designs such as FSATS, this subgraph is connected and has no transitions to *escape*. This is because the subgraph captures delays due to message passing before all actors enter an extension layer. In such cases, it can be used to sequentially compose a base system and an extension. Every state of the subgraph that contains exit state $exit_i$ enables transitions to in_i. Details appear in the full technical report.

Given the parallel composition of the extension machines constructed using the exit subgraph, we can attempt to verify the layer property using the environment model to generate the trains. This effort fails. The inbound operator sends the *two-in* message as soon as the environment model sends the*first* train into the tunnel; this is wrong, however, because the inbound operator should only enter the multiple train state when the *second* train enters the tunnel before the first train exits. This happens because some history between the *in* states and the environment is lost. Specifically, the environment model must have the first train in the tunnel and the second train approaching the tunnel at the *in* states of the extension; the normal environment model starts with both trains approaching the tunnel. We can synchronize the environment model with the extension by composing the environment model with the base system before computing the exit subgraph. The initial states of the exit subgraph now contain states of the environment model; those states should be used as the initial states of the environment when verifying properties of the layer. This construction indicates that the tunnel environment should start with the first train already in the tunnel.

Although generating restricted initial states of the environment model appears to be an overhead of formal verification, the problem of generating these models is similar to the problem of generating a testing harness for a layered design. Layered designs offer the hope of testing layers in isolation. That testing, however, requires knowledge about the environment that will drive the layer. Our approach merely formalizes the problem of obtaining a restricted testing harness for layered designs. In FSATS, the environment model problem arises because each extension corresponds to a new type of mission which is initiated only if the environment has generated a target of a particular type.

3.5.3 Verifying Properties Compositionally

The preceding section identified two key issues in supporting verification of layers independently from their base systems: synchronizing initial states and restricting environment models. Applying both techniques allows us to verify that an extension satisfies a given property relative to an interface to a base system. This does not address our entire problem, however, as we still must characterize when the properties of the composition of this layer with a base system can be verified via sequential composition, rather than on the global composed state machine.

The algorithm for checking property preservation under sequential composition requires that all states that are reachable in the composed system are contained in one of the layers being composed. For compositional verification to work, we must characterize when the reachable states of the composed system are contained in the reachable states of the union of the base system and the extension. In the general case, the desired result seems unlikely. Just as the actors do not enter an extension simultaneously, they do not exit from the layer simultaneously. The asynchronous exits may create reachable states in the composed system that are not contained in the extension. Worse still, these states may lead to states that become reachable in the base system only after composition. Either case would break our proposed layered verification methodology.

Fortunately, the collaborative designs that we have studied, including FSATS, tend to have a characteristic that addresses this problem: the reentry states *eventually* synchronize after executing an extension. Thus, with appropriate modeling of the reentry states in the extension, the sequential composition of the base system and the extension could capture the full global state space, as required for layered verification. We capture this model with a reentry subgraph that is computed in similar fashion to the exit subgraph; transitions to reentry states enable transitions in the reentry subgraph. Using the exit and reentry subgraphs, we can offer a CTL characterization of the cases in which our methodology is insufficient. When our methodology does not suffice, we will have to check the properties in the full composed system.[1]

The constraints that indicate when a property preservation can be confirmed (using a similar strategy as in the single state machine case) in layered fashion are as follows:

1. The *escape* state in the exit subgraph is not reachable under the restricted environment model.

2. The *reentry* states eventually synchronize: that is, once one layer machine reaches its reentry state, it remains there until all layer machines have reached their reentry states. This constraint is easily expressed as a series of CTL formulas to check of the model, one for each state machine in the extension:
$$\mathsf{AG}\ (reentry_i \rightarrow \mathsf{A}[reentry_i\ \mathsf{U}\ reentry_1 \wedge \ldots \wedge reentry_k])$$

We omit the proof that these conditions are sufficient to prove a correspondence between the two constructions of the global composed state space due to space constraints; the technical report will contain the full details. Intuitively, the proof consists of an argument that, under the above constraints, all reachable states under the first construction are reachable states in either the extension or the base system under the second construction. The interesting cases of this proof involve global states with some components in the base system and some in the extension. The conditions listed above restrict all such states to lie in the extension including the exit subgraph.

3.6 Implementation

We have conducted all the model checking tasks described in this paper. For this, we used the symbolic model checker VIS [32]. We modified VIS slightly to display all sub-formulas of properties generated during the marking phases; we used these sub-formulas for verifying the preservation of properties in other layers. For the paper's examples, the time and space usage are negligible.

Section 3 describes how we simulated the modular verification scenario while in fact attaching extensions to, potentially, the entire base system. This is because existing model checkers do not appear to be designed for extension to verifying open systems. For instance, they do not provide a way to query and assert properties on specific states. Expressing our extension layers in Verilog (VIS's input language) required manual insertion of additional design variables because we could not easily unify states in the underlying symbolic transition system. Finally, building the exit and reentry subgraphs was difficult in VIS's symbolic framework. Computing the core subgraphs is straightforward (by adding routines to the VIS source code); adding the escape state is difficult because it requires us to essentially reverse-engineer the symbolic state encoding to find an unused boolean representation for the escape state. A front-end for supporting layered design languages could work around the limitations of Verilog, but the limitations of the symbolic framework are harder to surmount.

Some properties can be verified in layers without the full power of model checking. For instance, simple properties that ensure a system always reaches a consistent state may not need extensive verification in an extension: simply showing reachability between the extension's *in* and *out* states often suffices (this relates to checking the requirement in our formal model that extensions yield connected graphs). These properties arise both in the examples presented in this paper and in FSATS. Therefore, there is clearly potential for applying more light-weight verification tools.

4 Conclusion and Future Work

Layered designs, which concentrate on the step-wise refinement of a system's features, offer an alternative to traditional object-based designs. They have arisen in several contexts, methodologies and applications, and appear to

[1] When we need to verify in the full composed system, we can apply existing techniques for parallel composition. As these techniques can be very difficult to use in practice, applying them effectively remains an open problem.

be especially promising in the context of software product-lines. Layered approaches share many of the software construction advantages of more traditional components.

This paper has explored how layered software designs require a different form of modular verification. We demonstrated that object-based decompositions of systems into modules that are concurrently or sequentially composed are inappropriate for layered designs. We also showed that layered, feature-based designs are actually quasi-sequential compositions of parallel compositions, and explained how certain constraints can make their verification tractable. We believe these constraints are reasonable because many applications appear to satisfy them. The resulting verification methodology minimizes the work expended to verify compositions relative to the work done verifying individual layers.

We have concentrated solely on model checking because we want to understand the strengths and limitations of algorithmic verification on layered designs. Our experience suggests that extant model checkers have not been designed to be extended for such tasks. (Certainly, a custom model checker is necessary to complete the verification of the entire suite of FSATS layers.) A related question is how to extend our approach to handle LTL formulas; for technical reasons, we have only considered CTL properties.

Layered designs can benefit from a broader scope of verification techniques. Early work on dependencies between layers [3] must be formalized and incorporated into any validation framework. We have encountered some layered designs involving complex data invariants that will likely be more amenable to theorem proving. While model checking captures and can verify the salient properties of the FSATS suite, it can be overkill. Preserving certain properties requires only simple results such as freedom from livelock or non-modification of particular variables. In these cases, simpler tools such as reachability engines and type systems may suffice. We expect further work with a richer set of designs to help us identify when the full power of our current methodology is required.

References

[1] Alur, R., R. Grosu and M. McDougall. Efficient reachability analysis of hierarchic reactive machines. In *International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2000.

[2] Alur, R. and M. Yannakakis. Model checking of hierarchical state machines. In *Symposium on the Foundations of Software Engineering*, pages 175–188, 1998.

[3] Batory, D. and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feburary 1997.

[4] Batory, D., C. Johnson, B. MacDonald and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, June 2000.

[5] Batory, D. and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[6] Biagioni, E., R. Harper, P. Lee and B. G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Symposium on Lisp and Functional Programming*, 1994.

[7] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992.

[8] Clarke, E., E. Emerson and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] Clarke, E., O. Grumberg and D. Peled. *Model Checking*. MIT Press, 2000.

[10] Clarke, E. M. and W. Heinle. Modular translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, August 2000.

[11] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zheng. Bandera : Extracting finite-state models from java source code. In *International Conference on Software Engineering*, 2000.

[12] Dwyer, M. B. and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report UM-CS-1999-052, University of Massachusetts, Computer Science Department, August 1999.

[13] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001. To appear.

[14] Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[15] Finkbeiner, B., Z. Manna and H. Sipma. Deductive verification of modular systems. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 239–275. Springer-Verlag, 1998.

[16] Fisler, K., S. Krishnamurthi and K. E. Gray. Implementing extensible theorem provers. In *International Conference on Theorem Proving in Higher-Order Logic: Emerging Trends*, Research Report, INRIA Sophia Antipolis, September 1999.

[17] Flatt, M. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, 1999.

[18] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[19] Grumberg, O. and D. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[20] Heineman, G. T. and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[21] Holzmann, G. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[22] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[23] Kupferman, O. and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[24] Laster, K. and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[25] McMillan, K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[26] Mezini, M. and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 97–116, October 1998.

[27] Pasareanu, C. S., M. B. Dwyer and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[28] Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[29] Steele, G. L., Jr., editor. *Common Lisp: the Language*. Digital Press, Bedford, MA, second edition, 1990.

[30] Stirewalt, K. and L. Dillon. A component-based approach to building formal-analysis tools. In *International Conference on Software Engineering*, 2001.

[31] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[32] The VIS Group. VIS: A system for verification and synthesis. In Alur, R. and T. Henzinger, editors, *International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, July 1996.

[33] van Renesse, R., K. Birman, M. Hayden, A. Vaysburd and D. Karr. Building adaptive systems using Ensemble. Technical Report 97-1638, Department of Computer Science, Cornell University, July 1997.

[34] VanHilst, M. and D. Notkin. Using role components to implement collaboration-based designs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 1996.

A Framework for Knowledge Management and Automated Constraint Monitoring

Ann Q. Gates and Steve Roach
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968
agates, sroach@cs.utep.edu

Abstract

This paper describes an approach called Dynamic Monitoring with Integrity Constraints (DynaMICs) that consists of a specification language for defining constraints and tools that permit automated instrumentation of constraints, runtime monitoring that minimizes performance degradation, and tracing. The goal is to capture domain and system knowledge as constraints and to use the constraints to monitor software execution, providing evidence of correctness and assistance in identification of error sources.

The paper presents a framework for managing knowledge and instrumenting programs to test the state of programs at runtime. In addition, it discusses the role that temporal logic, model checking and program-synthesis systems can play in developing and using DynaMICs.

1. Introduction

The successful development of complex software systems typically requires management of two types of knowledge: domain knowledge and system knowledge. Domain knowledge is that knowledge needed to understand problems and solutions in a particular area of human endeavor. For example, in order to construct a program that simulates wind tunnel experiments, system developers would need some knowledge of fluid dynamics.

System knowledge is that knowledge needed to understand the design and implementation of a software system that solves a problem from a particular domain. For example, in order to implement a module for an information management system, system developers would need to understand the structure of data to be processed and how it maps to the application domain. Domain knowledge is elicited from domain experts, customers, users, and possibly from written reference material. System knowledge comes from designers, implementers, and maintainers.

The integration of domain and system knowledge is essential for successful software development. This includes the capture and communication of such knowledge among team members throughout the software lifecycle. In addition, understanding the relations between domain and system knowledge can help close the gap between a software failure and the fault that leads to that failure.

One technique for capturing domain and system knowledge is through the use of integrity constraints. Integrity constraints, referred to as constraints in the remainder of this paper, are propositions about state of a system. State is defined as a set of program-variable and value pairs that captures a snapshot of memory during program execution. Constraints can be used to check during runtime that the system is meeting its requirements and that assumptions and limitations, which arise from design and implementation decisions, hold. Not only can constraints be used to identify errors and assist in debugging, they also can provide evidence of correctness at execution time, maintain knowledge about the domain and system, and assist in planning and implementing maintenance of the software.

This paper presents a framework for managing domain and system knowledge and for instrumenting programs to test the state of programs at runtime. The ultimate goal of this work is to automate the insertion of constraint-checking code in order to facilitate the use of constraints. With automated instrumentation, domain experts and system implementers may be more willing to expend effort generating the needed constraints to ensure correct execution.

1.1 DynaMICs

Dynamic Monitoring with Integrity Constraints (DynaMICs) is an approach that captures domain and system knowledge through constraints to ensure the correct functioning of a program during its execution (GT99, GT00, GT01). Fig. 1 presents a high-level view of the DynaMICs approach. The main features that differentiate DynaMICs from other software-fault monitoring approaches are as follows: constraint specifications are maintained in a repository separate from other artifacts, and constraint-checking code is automatically inserted into the code. The separation of constraint specifications from code facilitates identification of potential conflicts among constraints throughout the software's lifecycle. Algorithms translate propositions into constraint-checking code and determine the execution points at which the constraint-checking code is to be embedded into the program.

The tracing mechanism (GM01) provides support for establishing linkages between constraints and artifacts, which along with linkages that are created automatically during instrumentation, permits the following types of tracing: from application source code to constraint, from constraint to application source code, from constraint to requirements/artifacts, and from application code to requirements/artifacts. Because the links between the application code and constraints are created automatically during the instrumentation process, the links are maintainable. In addition, it eliminates the need for physic al links between artifacts and application code, which relieves the programmer from managing links in the code when code is revised, a tedious task that is prone to error. The approach addresses some of the issues that have slowed extensive adoption of tracing. Specifically, DynaMICs targets simplified tracing of constraints among documents, reduced overhead for tracing and, in conjunction with monitoring, requirements compliance with respect to constraints.

One of the reasons for the lack of widespread adoption of runtime monitoring is the performance degradation that results when constraint checks have been inserted into the program code. To address this, several different types of monitors are being investigated, e.g., one in which monitoring responsibilities are delegated to a process other than the one executing the application program (TM99, SM93).

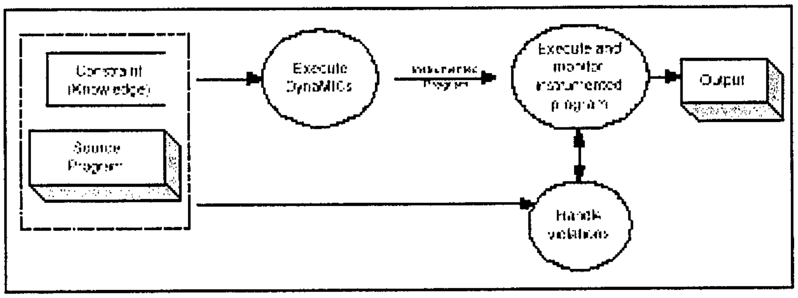


Figure 1: A high-level view of DynaMICs.

1.2 Impact of DynaMICs

Prediction of the impact of maintenance is difficult. In particula r, tools are needed to facilitate the identification of conflicts when new requirements or code are added. By managing domain and system knowledge, DynaMICs provides such support. Formally capturing knowledge allows reasoning about the effects of change and reduces the probability of creating errors in the software. The types of faults that are detectable through DynaMICs are dependent on the constraints specified by the development team. DynaMICs targets requirements faults, in particular those resulting from incorrect, inconsistent, and ambiguous requirements (GL98). In addition, DynaMICs is effective at detecting when an unpredictable sequence of events results in inconsistent data in the system (C01) and when the context of the program's operation changes as described in (LC95).

Communicating information among team members and eliminating conflicts in requirements are major concerns during development of complex systems (CK88, DV93). While successful software development efforts typically include at least one person who can integrate different perspectives on the development process as well as domain and system knowledge, there is an inherent risk in such dependence on key personnel. In addition, there is a limit as to how much knowledge can be managed by a single person, especially as systems become larger, more complex, and cross application domains. The capture of critical knowledge through constraints and the separation of constraints from the source program assist information management and communication.

Automated instrumentation of program code provides assurance of constraint coverage, i.e., DynaMICs determines the points in the program at which constraints from the repository should be checked. It also simplifies maintenance of constraints because changes are made to high-level constraint specifications and not to the code itself. This approach reduces the number of errors that can be introduced during maintenance because it is not possible for insertion or deletion of program code to corrupt previously inserted monitoring code. By establishing relations between constraint specifications and supporting documentation, it is also possible to provide justification of constraints. This supports resolution of conflicts between specifications and code.

An area in which this work can have impact is in creation and execution of test beds. Adding constraints during testing typically requires manual instrumentation of checking code and removal of the code when the product is deployed. Additionally, because of the large increase in code size due to instrumentation, execution times are naturally increased. By automating the insertion of constraint-checking code, developers are free from the task of instrumentation and can focus instead on improving the knowledge base that indicates correct program execution. Defects can be identified earlier in the testing cycle, reducing cost.

1.3 Organization of Paper

The remaining sections in this paper examine the constraint specification and automated instrumentation components of DynaMICs in which formal methods are critical. The main points covered in each section are as follows:

- overview of component, including a description of the knowledge needed, where it is obtained, and how its used;

- approach used to realize the component, including a discussion of the technologies that have been used to develop a proof-of-concept; and

- integration of formal methods, including a discussion of formal methods that impact development of a runtime monitoring system such as DynaMICs.

2. Constraint Specification

2.1 Overview

Initial work on DynaMICs has focused mainly on capturing domain knowledge and limited system knowledge through constraints. Domain knowledge includes properties, behaviors, and relationships among real-world objects being modeled by the software. System knowledge includes assumptions made by developers and limitations imposed by the system design. To have a larger impact on system development and maintenance, the knowledge base of DynaMICs must be extended to include additional system knowledge, in particular design knowledge derived from objects, data structures, operations, relationships among operations, and algorithms. Nevertheless, constraint-checking code establishes an implicit relationship between domain knowledge and the system.

Domain experts, clients, users, and members of the development team contribute to constraint definition. Other sources of constraints include documentation such as interview transcripts, memoranda, reports, and the requirements specification (GK97, DV93). Members of the development team contribute constraints by applying assumptions about the operating environment (e.g., acceptable input values) and limitations imposed by the design (e.g., size of a data structure). Testers, who are interested in monitoring program behavior under specific testing conditions, can add special-purpose constraints.

Constraint elicitation and identification necessitates analysis of the problem from a perspective different than requirements analysis. Requirements answer the question, "What will the system do?" while constraints answer the question "What monitored relationships can indicate correct program execution?". For example, consider the division of two integers, *a* and *b*, that yields a quotient, *q*, and a remainder, *r*. For this problem, two constraints can be defined:

$$r < b \text{ and } (q \times b) + r = a.$$

The constraints do not recalculate the division of *a* and *b*, rather they check that the division is correct. Domain (division) expertise is required to specify constraints such as these. A more detailed description of the constraint-definition process can be found in (GM01).

2.2 Approach

Constraints specifications consist of three parts: events, conditions, and actions. The event directs instrumentation by specifying the state at which the condition must hold. The condition expresses the constraint, and the action specifies what must be performed on violation of the condition. Each is discussed in the subsections that follow.

Constraints are captured during the requirements elicitation, requirements analysis, design, and implementation phases of software development. As a result, it is necessary to maintain the mapping from terms in the constraint specification to variables and storage locations at the program-code level. This is done through a data dictionary that maintains information about variables used in specifications, called *constraint variables*. Program development personnel maintain the associations of the constraint variables to program variables during program construction.

2.2.1 Event Specification

The event definition directs program instrumentation and is defined as an ordered five-tuple (GT99):

Event : Variable-set x States x Transition x Phase x Placement

Variable-set:	set-of-tokens
States	: {static, transitional}
Transition	: {immediate, intermediate, delayed}
Phase	: {input, processing, output}
Placement	: {before-store, after-store}

Events are based on stores to variables associated with constraint variables. *Variable-set* maintains the set of constraint-variable names for which *state transitions* (a change in the values held in constraint variables) are observed. *States* indicates the number of states needed to compute the value of the constraint. *Static* indicates that only the current (or anticipated) state is examined, requiring no instrumentation to save state. *Transitional* indicates that current and previous states are examined.

Variable -set, Transition, and *Phase* identify the state transitions to be monitored with respect to a specified phase. Valid phases include *input, processing,* and *output.* For a specified phase and *Variable-set, immediate* indicates that the constraint must hold after each state transition. *Delayed* denotes that a constraint must hold at the end of a specified phase. For example, a *delayed-on-input* constraint for variables *a, b,* and *c* indicates that the associated monitoring code executes after all values for these variables have been read. If a program uses an iterative construct to read in these values, then the check will occur at the point where the iterative construct terminates.

In the case of nested iterative constructs, the check will occur outside the outermost construct. For the specified phase and *Variable-set*, an *intermediate* value designates that the constraint must hold after an implied sequence of state transitions. For example, if *Variable-set* contains variables *a* and *b* and both variables are updated in a sequence, then the monitoring code executes after the sequence completes.

The types of instructions that cause state transitions within a program and that can be used to determine potential points of program instrumentation include input, assignment, and output instructions, i.e., instructions that store data to memory. Each is associated with a computation phase. Because computed values may not be stored to memory, but instead stored to a device via output instructions, it also is necessary to consider output instructions as instructions that cause state transitions. An assumption is made that controls of sensitive devices are memory mapped and, thus, are accessed using assignments. Constraints on file output, with a format that is clearly specified, can be checked (GT01).

Placement indicates whether a constraint is placed before a store or after a store to a constraint variable. In the case of a constraint in which a violation will cause catastrophic failure (referred to as a *mission-critical* constraint) it is imperative that the constraint is checked before the value is stored. A *critical* constraint is one in which a violation of the constraint indicates a hazard condition that could result in catastrophic failure. Constraints for critical and non-critical constraints can be checked after a store to a variable.

2.2.2 Condition Specification

The condition definition is expressed in a first-order language (G96). In addition to specifying relationships between program variables, conditions can determine, for example, whether the following hold: the value associated with a program variable is within a range of values, the value is a member of a set of values, a program variable has been assigned a value (not-null property), two sets of values are disjoint, and a set is a subset of another. Although we are using the term "program variables" in this discussion, it is important to note that the main use of constraints is to capture domain knowledge about the objects being modeled by the program. Because this type of constraint is implementation independent, a data dictionary, as described earlier, maps variables used in the constraint to model these objects to program variables.

2.2.3 Action Specification

The action specification defines the consequence of a constraint violation. This can include such actions as recording state in a history log, saving state for error recovery, performing state rollback, and initiating graceful degradation. The latter three actions are currently under study.

2.3 Integration of Formal Methods

Some practitioners may find expressing constraints in a formal temporal logic to be daunting. We believe that our specification language is more intuitive; however, it is semantically equivalent to a formal temporal logic. Using the language defined in (DV93) event *E* and condition *C* in a DynaMICs specification can be expressed as: (Es → Os C), where *Es* denotes an event that occurs in state *s*, *Os C* denotes that condition *C* holds in the state immediately following *s*, and *P* denotes that *P* holds in the current and all future states. Clearly, the advantage of a formal language is the ability to reason about constraints. This is essential for dealing with inconsistencies, one of the main software-development problems addressed by DynaMICs.

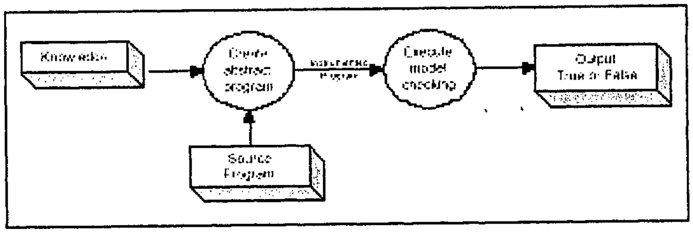


Figure 2: A high-level view of model checking.

The goals of runtime constraint checking are similar to model checking (H97). Model heckers monitor constraints of programs using an exhaustive finite-state search of an abstracted program (see Fig.2). Because the state space of complex programs is too large for exhaustive search, a common approach is to abstract the program to a model and test properties of the model. DynaMICs extends the range of model checking by allowing verification of constraints over a larger state space.

The approach provides runtime assurance that a property holds in a particular execution or test suite, i.e., it explores only a subset of the state space that a program visits and not the entire state space. Model checking will detect a failure if an error exists in the model. On the other hand, DynaMICs will detect a failure if an error exists in the program on a tested execution path. Model checking verifies a subset of a real program, and DynaMICs verifies a subset of states of a program.

One of the difficulties in model checking is the correct construction and instrumentation of a model from a program. We would like to investigate the possibility of automating model instrumentation for verification via model checking. Another area of research is the vertical traceability of errors detected by a model checker to code and specifications. One possible approach is to identify a failure using a model checker, instrument a source program using DynaMICs, then execute the instrumented source code using the state trace from the model checker to build a test suite. DynaMICs has the capability to trace to artifacts upon constraint violations through the tracing mechanism.

3. Instrumentation

3.1 Overview

Monitoring code is code that results in the test of a constraint. This can be an in-line sequence of instructions, a call to a procedure or function, or a trigger for an instruction sequence executed in a separate process (F98, L95). Identifying the points in program execution at which monitoring code should be executed requires the event definition of a specification along with analysis of the program's control flow. This section examines control-flow graph analysis and code generation.

3.2 Approach

3.2.1 Control-Flow Analysis

Analysis of a program's execution flow is needed in order to automatically determine program instrumentation points from constraint specifications (GP01). These points are associated with updates to monitored variables. The analysis approach followed by DynaMICS focuses on *path expressions* (T81, BM93, K97), i.e., regular expressions derivable from execution control-flow graphs (CFGs), each node of which is a basic block. A *basic block* (AS86) is a sequence of instructions with a single entry and single exit. Read/write lists are associated with each basic block to identify basic blocks of interest, i.e., ones that include accesses to monitored variables. Once these basic blocks are identified, the path expressions can be condensed by coalescing adjacent basic blocks.

Using the event definition as a guide, path expressions are analyzed to identify program instrumentation points; each of these points is associated with a unique *path tag* (TM99). Each path tag maps to a constraint specification. An algorithm for defining path tags for immediate, delayed-on-input, delayed-on-processing, and delayed-on-output constraints can be found in (F98, GT99, GP01). Checking immediate constraints is straightforward; the constraint check is performed whenever a monitored variable is modified. Placement of a delayed constraint requires identifying the best location at which to place a constraint check. For a delayed-on-process constraint, this is the point where assignment of the monitored variables is complete.

Analysis can be done at the intermediate-code or object-code level. In the case of mission-critical constraints, object-code analysis is needed to prevent a transition to an unsafe state. In memory-mapped IO systems, it may be necessary to prevent a write to memory prior to testing a constraint. For constraints that are not critical, intermediate-code instrumentation is sufficient. Because safety-critical systems require assurance that is not provided by current compilers, checking of critical and mission-critical constraints requires instrumentation at the object-code level.

3.2.2 Code Generation

One goal of DynaMICs is to synthesize constraint-checking code automatically from constraint specifications. Constraints may require information that is not computed by the program. This includes information that can be inferred from state and computed using counters and accumulators. The collection of this information is defined using an event-condition-action specification, where the event and condition are the same as a constraint specification. The main difference is that the action, which computes values to be used by constraints, is triggered when the condition is satisfied.

Consider a constraint that requires that a be less than b after a specified number of updates to b. In this case, variable, c is introduced to maintain the count of the number of updates to b. The event is an assignment to b (immediate-on-processing), the condition is *true,* and the action is the increment of c. The constraint can be specified as $c > threshold$ ® $a < b$ and is classified as an immediate-on-processing constraint on variable b.

Two classes of code are considered: *constraint-checking code* which ensures that the monitored program is executing correctly, as defined by constraints; and *information-generating code* which computes additional information needed to check constraints. Only the constraint-checking code raises violations for handling by the monitor. Neither will alter execution of the program except in the case when a violation is detected by constraint-checking code and the corresponding action definition requires error recovery. The information collected by information-generating code does not affect program execution since the variables used to maintain the information cannot be referenced in the source code.

Because constraint specifications and domain knowledge are implementation independent, the code-generation algorithm needs to translate specifications to code, considering possible differences between constraint-variable data types and associated program-variable data types stored in the data dictionary.

3.3 Integration of Formal Methods

Examples of program-synthesis systems that generate concrete -level code from abstract-level specifications are known (S91, SW94, SM96, W99). Some of these systems, in particular the fully automated deductive systems, suffer from their dependence on automated-theorem-proving tools.

However, it is possible that most of the constraints derived in practice will be classified by their structure (G96). In this case, it might be possible to take advantage of proof planning or schema-guided synthesis (WB92, BS90). Experiments conducted thus far have shown that the code to test a constraint is reasonably small, on the order of tens of lines of high-level code (C01). This small size of code enables the use of fully automated tools.

The aforementioned approaches require complete axiomatization of domain knowledge. If such knowledge is not available, inductive techniques for the synthesis of constraint-checking code could be useful (F95, MB98). This would allow the collection of examples from which the system will be able to generalize.

4. Summary

The difficulty in communicating crucial knowledge among team members from specification to software deployment, managing change, and formally specifying requirements makes verification of software challenging. To address this, it is imperative that developers begin to focus on identifying pertinent domain and system knowledge that can be used to determine whether a program is operating correctly during its execution. Successful software projects have key personnel who can integrate several knowledge domains as well as system knowledge. The aim of the work presented in this paper is to capture such knowledge as constraints and to use the constraints to monitor the program during runtime.

This will facilitate identification of errors by closing the gap between a software failure and the fault that leads to failure. The focus of this paper is to describe a framework for an approach called DynaMICs that assists in providing evidence of correctness in software systems and assistance in identification of error sources. Additionally, the paper describes the practical impact of formal methods on development of DynaMICs.

The DynaMICs approach differs from other monitoring approaches because constraints are stored separately from other artifacts and instrumentation of constraint-checking code is automated. We believe that the automation provided by the approach, the ability to monitor correct behavior of programs, and the ability to trace to artifacts will motivate the capture and use of constraints. Because the formal language used by DynaMICs is semantically equivalent to a formal temporal logic, a tool that supports reasoning about constraints and detection of potential inconsistencies in requirements will make the approach even more attractive.

The goals of DynaMICs are complementary to model checking. The automated instrumentation algorithms of DynaMICs may be applicable to instrumentation of abstract programs in model checkers, and model checkers can support the use of DynaMICs. For example, model checkers can direct creation of test suites from state traces upon failures and, using these test suites, DynaMICs can facilitate the identification of faults in the program and assist in error resolution through the tracing mechanism.

In order for DynaMICs to be applicable to high-assurance systems, it is crucial to generate provably correct, executable code. Generation of constraint-checking code from specifications is a good candidate for application of program-synthesis systems that take advantage of proof-planning or schema-guided synthesis. The main reasons are the simple and concise pieces of code that are generated from each constraint and the fact that constraints can be classified based on their structure.

References

[AS86] Aho, A., Sethi, R. and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[BM93] A. Bertolino and M. Marré, "Deriving Path Expressions Recursively," *IEEE Second Workshop on Program Comprehension*, pp. 177-185, July 1993.

[BS90] A. Bundy, A. Smaill, and G. Wiggins, "The Synthesis of Logic Programs from Inductive Proofs," in J. W. Loyd (ed), *Proceedings of the ESPRIT Symposium on Computational Logic*, pp. 135-149, Springer-Verlag, 1990.

[C01] R. Cereceres, "A Study Of The Effectiveness Of Integrity Constraints in the DynaMICs Approach", Master's Project Report, The University of Texas at El Paso, El Paso, Texas, 2001.

[CK88] B. Curtis, H. Krasner, and N. Iscoe., "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM*, 31(11), pp. 1268-1287, 1988.

[DV93] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquistion", *Science of Computer Programming*, 20, pp. 3-50, 1993.

[F98] F. Fernandez, "Compiler-driven Approach to Monitoring Integrity Constraints", Master's Thesis, The University of Texas at El Paso, El Paso, Texas, 1998.

[F95] P. Flener, "Logic Program Synthesis from Incomplete Information", Klewer Academic Publishers, Norwell, MA, 1995.

[G96] A. Gates, "On Defining a Class of Integrity Constraints", *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering*, pp. 338-344, 1996.

[GK97] A. Gates and C. Kubo Della -Piana, "The Identification of Integrity Constraints in Requirements for Context Monitoring", *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems*, pp. 498-505, 1997.

[GL98] A. Gates and S. Li, "Software Faults and their Detection through DynaMICs," *Proceedings of the IASTED Conference Software Engineering*, Las Vegas, NV, pp. 323-327, 1998.

[GT99] A. Q. Gates and P. J. Teller, "DynaMICs: An Automated and Independent Software-Fault Detection Approach", *Proceedings of the Fourth International High-Assurance Systems Engineering Symposium,* pp. 11-19,1999.

[GT00] A. Q.Gates and P. J. Teller, "An Integrated Design of a Dynamic Software-Fault Monitoring System", *Journal of Integrated Design & Process Science. Society for Design and Process Science*, 14(3), 63-78, 2000.

[GT01] A. Q. Gates and P. J. Teller, "Dynamic Software Monitoring with Integrity Constraints: A Unified Approach for the Development and Evolution of Software", in review, 2000.

[GP01] A. Q. Gates, O. Mondragon, S. Roach, and A. Provetti, "Object-level Constraint Instrumentation: From Control Flowgraphs to Path Expressions", submitted to The Eighth International Static Analysis Symposium, February 2001.

[GM01] A. Q. Gates and O. Mondragon, "A Constraint-Based Tracing Approach", to appear *Journal of Systems and Software,* 2001.

[H97] G. Holtzman, "The Spin Model Checker," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

[K97] M. Kidd, "Ensuring Critical Event Sequences in High Consequence Computer Based Systems as Inspired by Path Expressions," *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems*, pp. 483-490, 1997

[L95] J. R. Larus, "EEL: Machine-Independent Executable Editing", *SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* June 1995.

[LC95] Luqi and D. Cooke, "How to Combine Nonmonotonic Logic and Rapid Prototyping to Maintain Software," *International Journal on Software Engineering and Knowledge Engineering*, 5(1), pp. 89-118, 1995.

[MB98] S. Muggleton and W. Buntine, Machine invention of first-order predicates by inverting resolution," *Proceedings of the 1988 International Conference on Machine Learning*, pp. 339-352, 1988.

[SM93] S. Sankar and M. Mandal, "Concurrent Runtime Monitoring of Formally Specified Programs," *IEEE Computer*, 26(3), pp. 32-41, 1993.

[S91] D. R. Smith, "KIDS: A Knowledge-Based Software Development System", in *Automating Software Design*, M. Lowry and R. McCartney (eds.), MIT Press, pp. 483-514, 1991.

[SM93] Y. V. Srinivas and J. L. McDonald, "The Architecture of Specware, a Formal Software Development System", Kestrel Institute Technical Report KES.U.96.7, 1996.

[SW94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive Composition of Astronomical Software from Subroutine Libraries", *Proceedings of the 12th Conference on Automated Deduction*, Nancy, France, June 28-July 1, 1994.

[T81] Tarjan, R.E., “Fast Algorithms for Solving Paths Problems,” *Journal of the ACM* , 28(3), pp 584-614, 1981.

[TM99] P. Teller, M. Maxwell, and A. Gates, “Towards the Design of a Snoopy Coprocessor for Dynamic Software-Fault Detection”,” *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, pp. 310-317, February 1999.

[WB92] G. Wiggins, A. Bundy, I. Kraan, and J. Hesketh, “Synthesis and Transformation of Logic Programs from Constructive, Inductive Proof,” *Proceedings of LOPSTR'91*, Springer-Verlag, pp. 27-45, 1992.

[W99] V. Winter, “An Overview of HATS: A Language Independent High Assurance Transformation System”, Sandia National Laboratories, 1999.

The Use of Computer Aided Prototyping for Re-engineering Legacy Software[1]

Luqi, V. Berzins, M. Shing
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

Abstract

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. The process of re-engineering old procedural software to a modern object-oriented architecture introduces certain complexities into the software analysis process. The direct products of reverse engineering, such as requirements or design specifications, are likely to have a functionally based structure. As a result, some transformation of the recovered requirements and design specifications is necessary in order to obtain specifications for the new structures. It is often very difficult to quickly determine if the transformed specification is a true representation of the desired requirements. This paper discusses the effective use of computer-aided prototyping techniques for re-engineering legacy software, and presents results of a case study which showed that prototyping can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed.

Keywords: Software re-engineering, Object-oriented architecture, Computer-aided prototyping, Software evolution, Combat simulation

1. Introduction

Legacy systems embody substantial institutional knowledge, which includes basic and refined requirements, design decisions, and invaluable advice and suggestions from domain users that have been implemented over the years. To effectively use these assets, it is important to employ a systematic strategy for continued evolution of the current system to meet the ever-changing mission, technology and user needs. Re-engineering has frequently been proven to be more cost effective than new development and is also known to better promote continuous software evolution.

However, the institutional knowledge implicit in a legacy system is difficult to recover after many years of operation, evolution, and personnel change. These software systems were originally written twenty or more years ago using what many now view archaic and ad-hoc methods. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that now must be changed over large areas of the code, and may have introduced inconsistencies and faults.

Software re-engineering can be defined as the systematic transformation of an existing system into a new form to realize quality improvements, such as increased or enhanced functionality, better maintainability, configurability, reusability, performance, or evolvability at a reduced cost, schedule, or risk to the customer. This process involves recovering existing software artifacts from the system and then transforming and re-organizing them as a basis for future evolution of the system. Since typical legacy systems were originally designed and implemented using a functionally based approach, some transformation of the recovered information is necessary in order to obtain an object-oriented model. It is often very difficult to obtain a transformed specification that accurately represents the desired requirements.

Since legacy systems are usually re-engineered only when the existing systems need some kind of improvement, it is unlikely that the initial version of the reconstructed requirements adequately reflects

[1] This research was supported in part by the U.S. Army Research Office under contract number 350367-MA and 40473-MA.

current user needs. Prototyping provides a means to identify and validate changes to system requirements while simultaneously enabling prospective users to get a feel for new aspects of the proposed system. It is a well-established approach that can be highly effective in increasing software quality [15]. When used in conjunction with conducting a major re-engineering effort, prototyping can be extremely useful in assisting in many areas of software modification, validation, risk reduction, and the refinement of new software architectures and user requirements.

This paper describes a case study that illustrates the effective use of computer-aided prototyping techniques for re-engineering legacy software [3, 16]. The case study consists of developing an object-oriented modular architecture for the existing US Army Janus(A) combat simulation system [19], and validating the architecture via an executable prototype using the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School [14]. Janus(A) is a software-based war game that simulates ground battles between up to six adversaries [9]. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of over 350,000 lines of FORTRAN code. The FORTRAN modules are organized as a flat structure and interconnected with one another via 129 FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create a base-line object-oriented architecture that supports existing and required enhancements to Janus functionality.

The paper presents the re-architecturing process and the resultant object-oriented architecture. in Sections 2. Section 3 describes the use of computer aided prototyping to validate the resultant architecture and Section 4 draws some conclusions.

2. The Re-Architecturing Process

The re-architecturing process used in the case study consists of 3 major phases: reverse engineering, object-oriented design and design validation via prototyping (Figure 1).

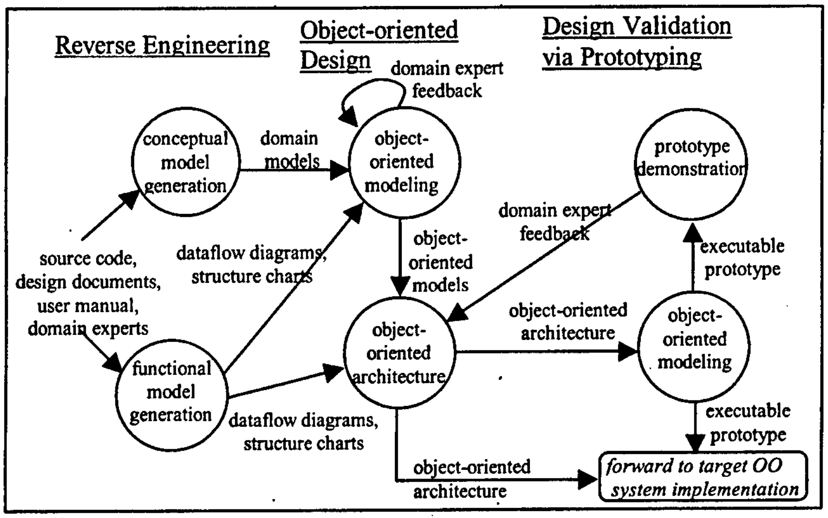

Reverse Engineering
Object-oriented Design
Design Validation via Prototyping
domain expert feedback
conceptual model generation
domain models
object-oriented modeling
prototype demonstration
domain expert feedback
source code, design documents, user manual, domain experts
dataflow diagrams, structure charts
object-oriented models
executable prototype
functional model generation
object-oriented architecture
object-oriented architecture
object-oriented modeling
dataflow diagrams, structure charts
executable prototype
object-oriented architecture
forward to target OO system implementation

Figure 1. The object-oriented re-architecturing process.

2.1 Reverse Engineering

The first phase is reverse engineering. Input to this phase includes the legacy source code, design documents, user manuals, and information from domain experts. Since the goal of the initial re-engineering effort is to duplicate the functionality of the existing system within a modular, extensible architecture and to reuse domain concepts, models and algorithms instead of the existing code, we should avoid including any requirements/constraints that are consequences of issues related to FORTRAN implementation. The best places to extract domain concepts from the existing system are the user manuals and the database management system manuals. These manuals were written using the lingo of the user community and should be relatively free of implementation details. We found the JANUS Data Base Management Program Manual [10] particularly useful because it contains detailed information on what kind of data are needed to model the battlefield and how they are organized (logically) in the database. The top-level structure of the database is shown in Figure 2.

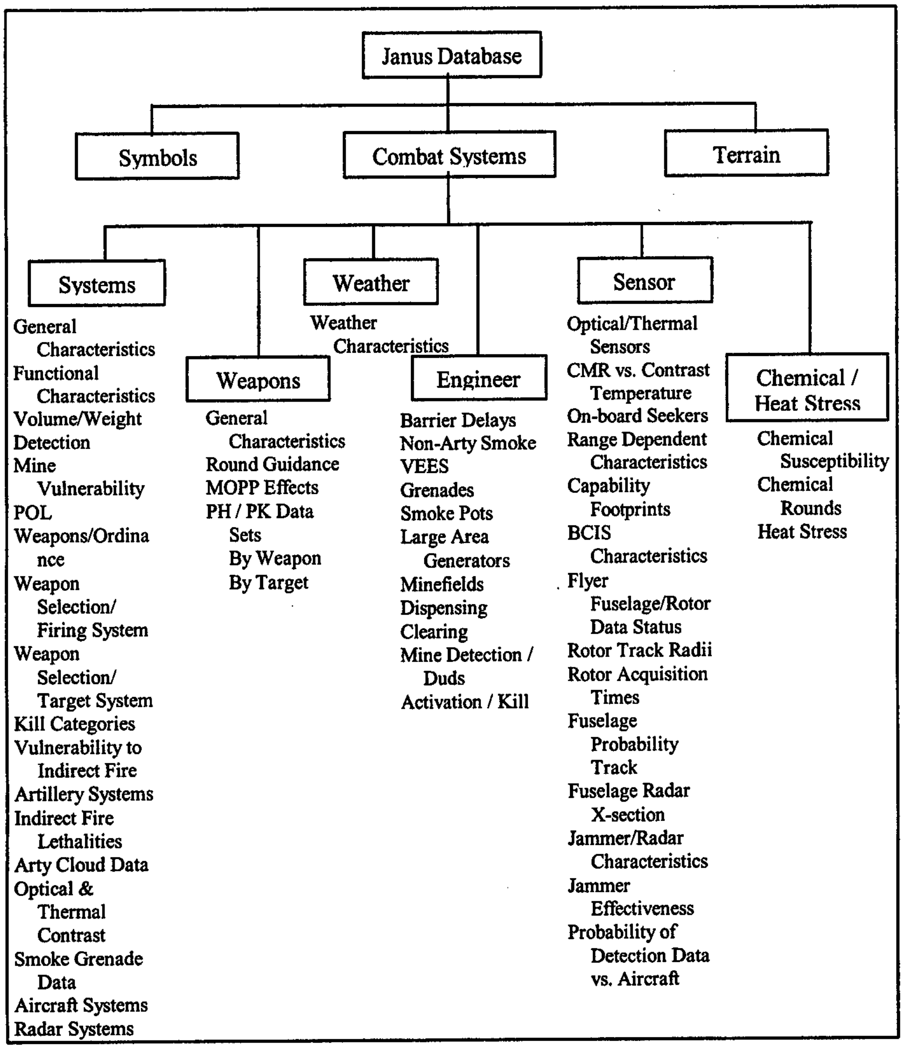


Figure 2. The top-level structure of the Janus Database.

Not shown in Figure 2 are the interdependencies between the data, whereby data entered in one category affect directly or indirectly the data in other categories. For example, the barrier delay attributes of the Engineer Data depend on specific weather conditions derived from the Weather Data and system functional characteristics derived from the System Data. The overall network of interdependencies is highly complex and can only be understood through construction and analysis of a functional model of the existing Janus software.

Analysis of the legacy implementation of 350,000 lines of source code is a daunting but inescapable part of this step. We recoiled from the magnitude of this effort and analyzed the Janus User's manual [9], the Janus Programmer's Manual [7], the Janus Software Design Manual [8], and the Janus Algorithm Document [18] instead. These documents helped us get started because they contained higher level information and were much shorter than the code. However, they were also older, and it was a constant struggle to determine which parts were still accurate, and which were not. In hindsight, avoiding analysis of the code was a mistake that slipped the schedule of the project by several months. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross-fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively.

Using manual techniques augmented with the text matching tool *grep* [1], which takes a regular expression and a list of files and lists the lines of those files that match the pattern, we were able to walk through the code and get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [7] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs and develop functional models from the data flows. We used CAPS to assist in developing the abstract models [3]. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

We also had a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. These meetings were indispensable because they gave us information that was not present in the code. Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [4]. Domain analysis has been identified as an effective technique for software re-engineering [17]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

2.2. Object-Oriented Design

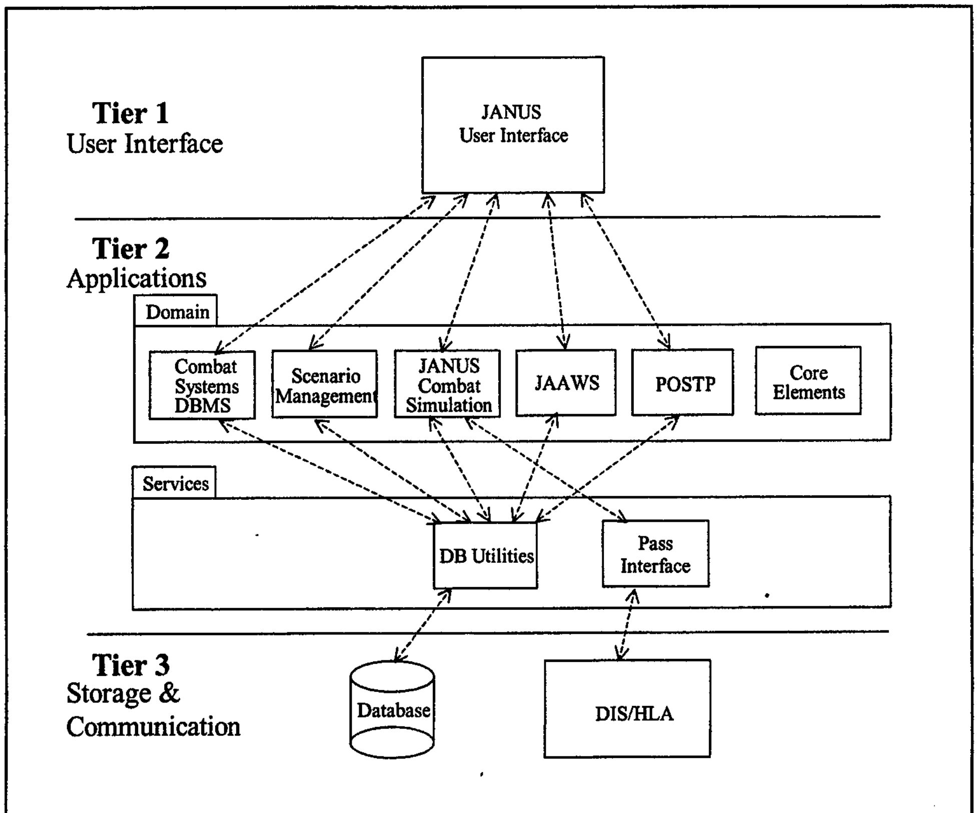
Next, we developed object models and architecture of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [5]. This was probably the most difficult and most important phase. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this phase, we used our knowledge of object-oriented analysis and the UML notations to create the classes and associated attributes and operations [20]. This was a crucial phase because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software.

Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [6]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

The re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core data elements and the object-oriented architecture for the Janus System. We presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center project. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced in [11] that the involvement of domain experts is critical for nontrivial re-engineering tasks.

Early involvement of the stakeholders in the simulation community also paid off in the long run. Both the National Simulation Center and Combat21 projects were able to save time and money by reusing our work and came up with designs that look remarkably like ours (although much larger). Now, OneSAF developers have been directed to look at the Combat21 class design and reuse as much as possible. So, our efforts have directly benefited other simulation developers.

Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 3).



Note: Lines showing the dependency of the Combat Systems DBMS, Scenario Management, JANUS Combat Simulation, JAAWS, POSTP, DB Utilities and Pass Interface subsystems on the Core Elements packages are omitted from the diagram to keep it clear and simple.

Figure 3. The resultant 3-tier object-oriented architecture.

We extracted most of the data and operations from the existing Combat System DBMS, Scenario Management, Janus Combat Simulation, JAAWS and POSTP subsystems and encapsulated them as simulation objects in the Core Elements package, leaving only application specific control codes that use the simulation objects in each of these five subsystems. Figures 4 and 5 show the top level class structures of the object models of the core elements. Details of the associated attributes and operations can be found in [3, 22] and are omitted from these diagrams due to space limitations.

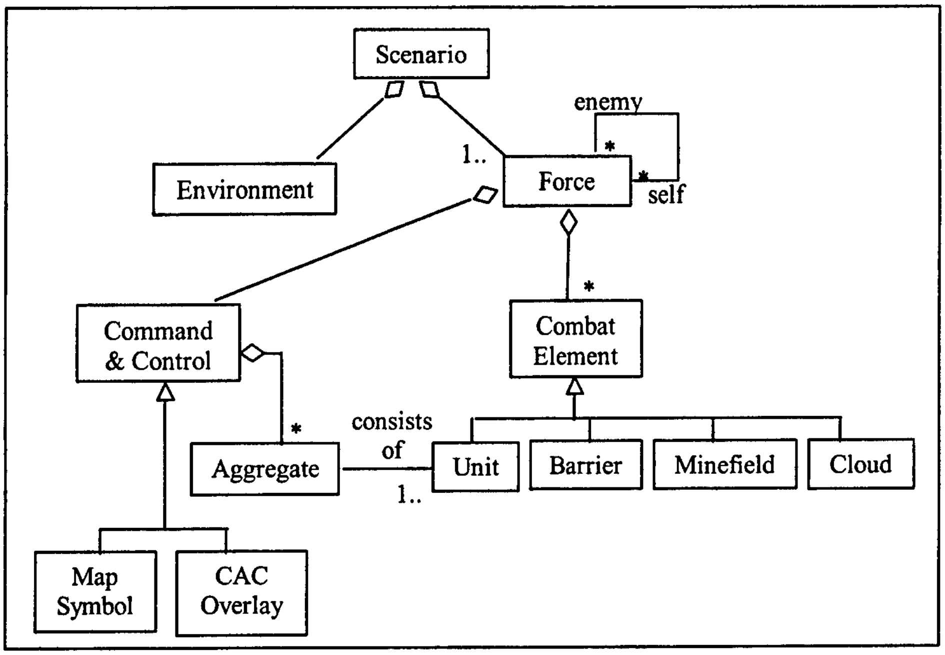


Figure 4. The top-level structure of the Janus Core Elements Object Model.

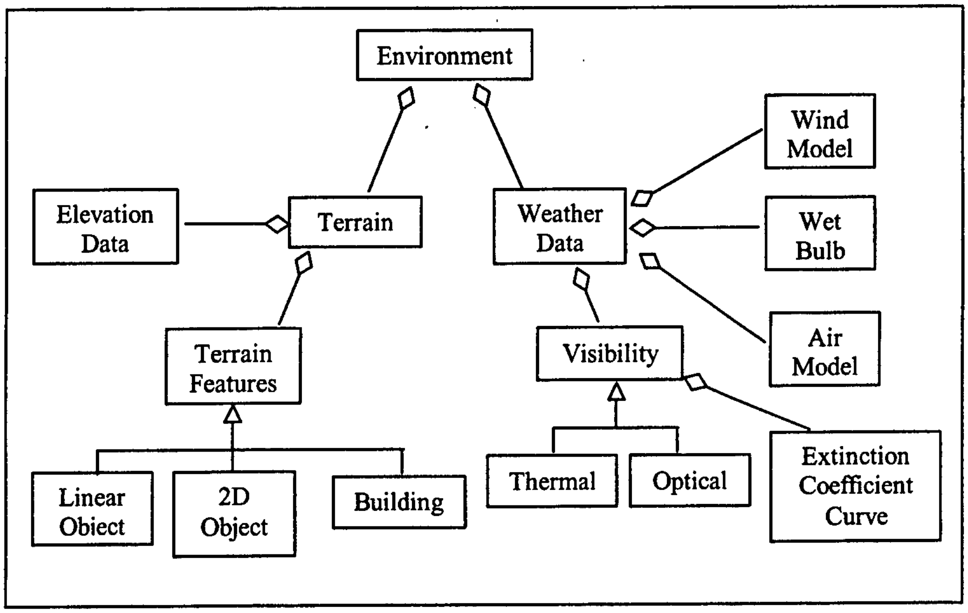


Figure 5. The Environment Object Class.

Central to the Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advance the game clock to the scheduled time of the event and perform that event. The existing Janus Simulation System uses 17 different categories to characterize the events. RUNJAN then handles these 17 events using the event handlers shown in Figure 6.

1) DOPLAN - Interactive Command and Control activities
2) MOVEMENT - Update unit positions
3) DOCLOUD - Create and update smoke and dust clouds
4) STATEWT - Periodic activity to write unit status to disk
5) RELOAD - Plan and execute the direct fire events
6) INTACT - Update the graphics displays
7) CNTRBAT - Detect artillery fire
8) SEARCH - Update target acquisitions, choose weapons against potential targets, and schedule potential direct fire events
9) DOCHEM - Create chemical clouds and transition units to different chemical states
10) FIRING - Evaluate direct fire round impacting and execute indirect fire missions
11) IMPACT - Evaluate and update the results of an indirect round impacting
12) RADAR - Update an air defense radar state and schedule direct fire events for "normal" radar
13) COPTER - Update helicopter states
14) DOARTY - Schedule indirect fire missions
15) DOHEAT - Update unit's heat status
16) DOCKPT - Activity to record automatic checkpoints
17) ENDJAN - Housekeeping activity to end the simulation

Figure 6. The event handlers for the legacy Janus system.

Like all typical Fortran programs, the existing event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation Subsystem was to distribute the event handling functions to individual objects. However, many of the current event handler categories contained redundant code. They did not seem to be independent of each other and were not consistent with the class hierarchy we created. For example, the set of event handlers used to simulate the activities of a particular unit to search for targets, select weapons, prepare for a direct fire engagement, and then execute that direct fire engagement differs depending upon whether the unit has a normal radar, special radar, or no radar at all. The existing Janus Simulation System uses the RADAR event handler to carry out the entire procedure if the unit has normal radar. However, it uses the SEARCH, RADAR, and RELOAD event handlers to carry out the procedure if the unit has special radar. Finally the system uses the SEARCH and RELOAD event handlers to conduct the procedure if the unit has no radar at all. We conjecture that this lack of uniformity is due to a series of software modifications made by different people at different times without full knowledge of the software structure. The example also illustrates another problem: the legacy event handlers were not designed to perform independent tasks, and had complicated interactions with each other.

It was necessary to redefine some event categories in order to reduce interdependencies between the event handlers, to factor simulation behavior into more coherent modules, to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Moreover, the Janus system was originally designed to work in isolation, and has since been adapted to interact with other simulation systems. Interactions between the simulation engine and the world modeler (the interface to the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation. The new architecture uses an explicit priority queue of event objects to schedule the simulation events. We were able to reduce the total number of event handlers needed in the simulation, from 17 to 14, by eliminating identified redundant code (Figure 7).

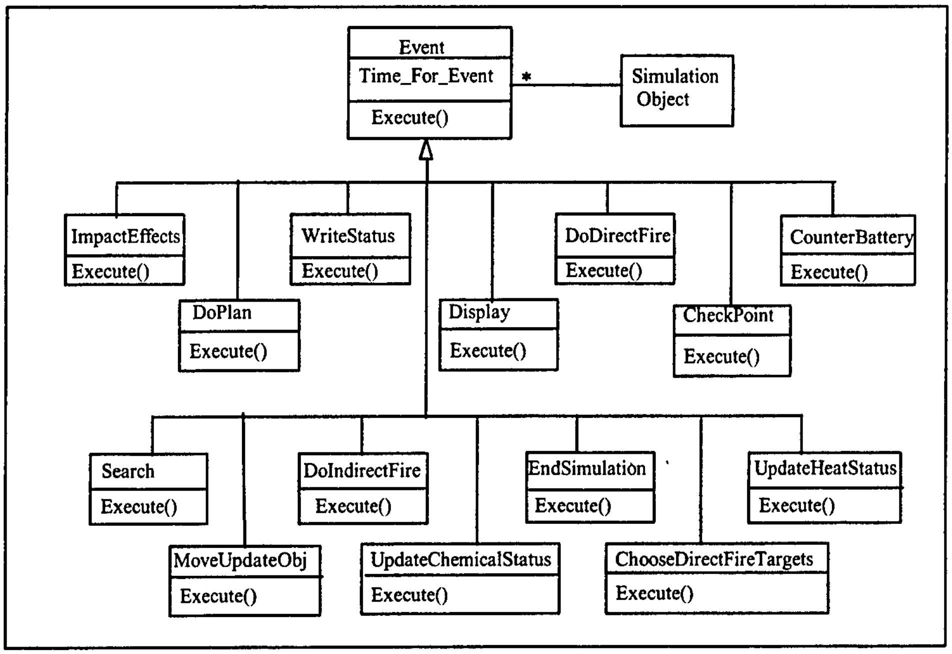


Figure 7. The event class hierarchy.

We tried to make the actions of the new event handlers independent and orthogonal. Independent means that one event handler does not invoke or depend on the action of another. Orthogonal means that the purpose of one event handler is completely separate from that of another. Although our architecture does not completely meet these goals, it comes much closer to them than the legacy design does. We believe that these properties of the architecture are desirable because they impose a partitioned structure on the system that aids future enhancements and modifications. If an enhancement affects only one kind of event, then it becomes relatively easy to isolate the affected part of the code. If suitable naming conventions are followed, relatively low-tech tool support will be adequate for helping system maintainers find the parts of the code that must be understood and modified to make a future change to the system.

Every event has an associated simulation object in the new architecture. This associated object is the target of the event. Depending on the subclass to which an event object belongs, the "execute" method of the event will invoke the corresponding event handler of the associated simulation object. (See [3] for details.) The new event hierarchy enables a very simple realization of the main simulation loop:

```
initialization;
while not_empty(event_queue) loop
    e := remove_event(event_queue);
    e.execute();
end loop;
finalization;
```

Note that this same code is used to handle all of the event handlers, including those for future extensions that have not yet been designed. Event objects with associated simulation objects are created and inserted into the event queue by the initialization procedure, the constructors of simulation objects, and the actions of other event handlers. Depending on the actual event, events are inserted into an event priority queue based on time and priority.

Our newly designed architecture eliminates the need for the simulation loop to know what kind of object it is handling. Thus when adding an object type not yet designed, the simulation loop does not require additional code to invoke the new object's event handlers. By localizing all changes to the newly added object class, our architecture eliminates the possibility of introducing errors into the existing parts of the simulation.

3. Design Validation Via Prototyping

The process of transforming a design developed using the functional approach into an object-oriented design introduces risks of unintentionally altering system behavior. In the context of our case study, the resultant object oriented architecture and the new event dispatching control structure are areas of high risk since they differ significantly from the functional design of the legacy software. UML provides two ways to model behavior. One is to capture the behavior of individual objects over time using state machines, and the other is to capture the interactions of a set of objects in the system using sequence diagrams and collaboration diagrams. While state machines are precise, they only focus on a single object at a time and is hard to understand the behavior of the system as a whole. The sequence diagrams and the collaboration diagrams, on the other hand, lack a formal semantics for precise description of the system behaviors.

One way to reduce the risk is to validate the dynamic behavior of the proposed architecture and to refine the interfaces of subsystems via prototyping at the early design stage. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Computer aid for constructing and modifying prototypes makes this feasible [15]. The CAPS system is an integrated set of software tools that generate source programs directly from high-level requirement specifications.

Due to time and resource limitations, we developed a prototype for only a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (*move*, *do_plan*, and *end_simulation*), and one kind of post-processing statistics (fuel consumption).

We developed an executable prototype using CAPS. Figure 8 shows the top-level structure of the prototype, which has four subsystems: *janus*, *gui*, *jaaws* and the *post_processor*. Among these four subsystems, the *janus* and the *gui* subsystems (depicted as double circles) are made up of sub-modules while the *jaaws* and the *post_processor* subsystems (depicted as single circles) are mapped directly to modules in the target language. After entering the prototype design into CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems. In addition, a simple user interface was developed using CAPS/TAE [21].

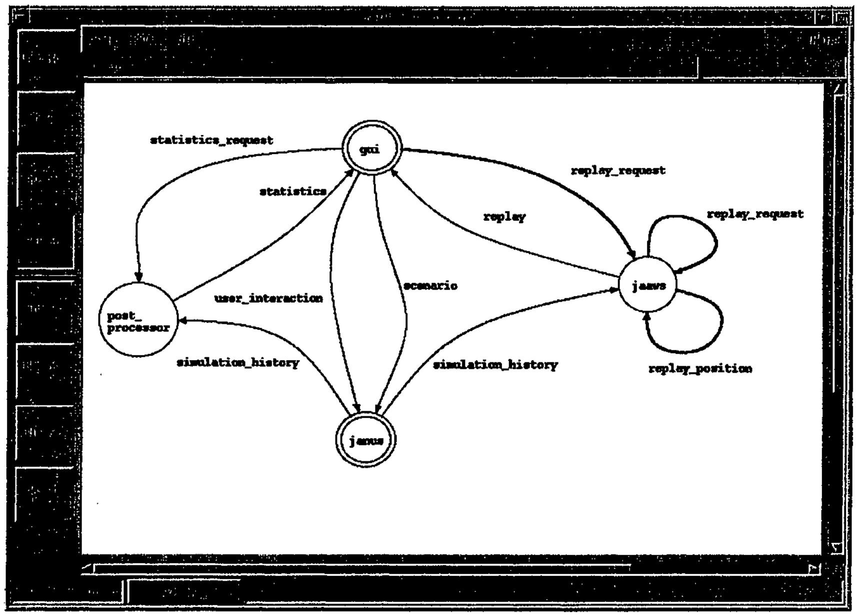


Figure 8. Top-level decomposition of the executable prototype.

The resultant prototype has over 6000 lines of program source code, most of which was automatically generated, and contains enough features to exercise all parts of the architecture. The code that handles the motion of a generic simulation object was very simple, but it was designed so that it would work in both two and three dimensions without modification (currently the initialization and the movement plan of the tank object never call for any vertical motion). The code was also designed to be polymorphic, just as was the main event loop. This means the same code will handle the motion of all kinds of simulation objects without any modifications, including new types of simulation objects that are part of currently unknown future enhancements to Janus and have not yet been designed or implemented.

Our prototyping experiment showed that the proposed object-oriented architecture allows design issues to be localized and provides easy means for future extensions. We started out with a prototype consisting of only two event subclasses (*move* and *end_simulation*) and were able to add a third event subclass (*do_plan*) to the prototype without modifying the event control loop of the Janus combat simulator.

We also demonstrated the use of inheritance and polymorphism to efficiently extend/specialize the behavior of combat units. For example, the *move_update_object* method of a tank subclass uses the general-purpose method from its superclass to compute its distance traveled and a specialized algorithm to compute its fuel consumption. We simply include one statement to invoke the *move_update_object* method of its superclass followed by three lines of code to update its fuel consumption. Moreover, other combat unit subclasses can be added easily to the prototype without the need to modify the event scheduling/dispatching code and usually without modifying existing event handlers.

The issues raised by the design of the prototype also resulted in the following refinements to the proposed architecture:

1. Extend the interface of the *Execute_Event* operation to return the time at which the next event is to be scheduled for the same simulation object, and introduce a special time value "NEVER" to indicate that no next event is needed. The proposed change turns the communication between the event dispatcher and the simulation objects from a peer-to-peer communication into a client-server communication. This change eliminates dependencies of event handlers on event queue details and allows the event dispatcher to use a single statement to schedule all recurring events for all event types.
2. Instead of recording the history of a simulation run in sets of data files, model the simulation history as a sequence of events. The proposed change provides a simple and uniform way to handle history records for all events, and allows the same modular architecture to be used for real-time simulations as well as post-simulation analysis. It also eliminates the need for the write-status event, reducing the number of events still further. This approach provides the greatest possible resolution for the event histories, which implies that any quantity that could have been calculated during the simulation can also be calculated by a post-simulation analysis of the event history, without any loss of accuracy. The only constraint imposed by this design refinement is that the simulation objects in the events must be copied before being included in the simulation history, to protect them from further changes of state as the simulation proceeds. This constraint is easy to meet in a full-scale implementation because the process of writing the contents of an event object to a history file will implicitly make the required copy.
3. It is beneficial to allow null events appear in the event queue. A null event is one that does not affect the state of the simulation, such as a move event for an object that is currently stationary. The prototype version adopted the position that such events should not be put in the event queue, since this corresponds to current scheduling policies in Janus, and appears at first glance to improve efficiency. Our experience with the development of the prototype suggests that this decision complicates the logic and may not in fact improve efficiency. The current design uses the process *create_new_events* to scan all simulation objects once per simulation cycle to determine if any dormant objects have become active, and if so, schedules events to handle their new activity. The alternative is to have the constructor of each kind of simulation object schedule all of its initial events, and to have each event handler specify the time of next instance of the same event even if there is nothing for it to do currently. Handlers might still set the time of its next event to NEVER in the case of a catastrophic kill; however this is reasonable only if it is impossible to repair or restore the operation of the units that have suffered a catastrophic kill. The reasons why this design change may improve efficiency in addition to simplifying the code are that:
   (a) the check for whether a dormant object has become active is done less often - once per activity of that object, rather than once per simulation cycle,
   (b) executing a null event is very fast - a few instructions at most, so the "unnecessary" null events will not have much impact on execution time, and
   (c) the computation to find and test all simulation objects periodically would be eliminated.

   We recommend allowing null events in the event queue, and explicitly scheduling every kind of event for every object unless it is known that there cannot be any non-empty events of that type in any possible future state of the object. For example, under the proposed scheduling policy, immobile or irrecoverably damaged objects would not need to schedule future move events, but those that are currently at their planned positions would need to do so, because a change of plan could cause them to move again in the future, even though they are not currently moving. The resulting architecture enables a very simple realization of the main simulation.

4. Conclusion

Our conclusion is that substantial and useful computer aid for re-engineering is possible at the current state of the art. Human analysts and domain experts must also play an important part of the process because much of the information needed to do a good job is not present in the software artifacts to be re-engineered. Success depends on cooperation between skilled people and appropriate software tools.

The missing information needed for re-engineering is related to deficiencies of the current system at all levels, from requirements through design and implementation. Thorough and accurate knowledge of these deficiencies is crucial for success. The clients never want the re-engineered system to have the exactly same behavior as the legacy system - if they were satisfied, there would be little motivation to spend time, effort, and resources on a re-engineering project. Even if a system is being re-engineered for the ostensible goal of porting to different hardware, the desired behavior at the interface to the hardware and systems software will be different.

In practical situations, the requirements for the re-engineered system are different from those for the legacy system. Key parts of the requirements for the new system are often missing or incorrect in the legacy documents. Some of that information is present only in the minds of the clients, often fragmented and scattered across members of many different organizations. Communication is a large part of the process, and that communication cannot be automated away, although it can be enhanced by appropriate use of prototyping. We found that the most important communications were those regarding newly recognized requirements issues, and that such recognition were often triggered by discussions between people with different areas of expertise.

Uncertainties about the true requirements play a central role in both re-engineering and the development of new systems. We therefore hypothesized that prototyping could play a valuable role in re-engineering efforts. Our experience in the case study reported here support that hypothesis.

We also found that prototyping can contribute substantially to the process of inventing, correcting, and refining the conceptual structures on which the architecture of the new system will be based. Most legacy systems are too complicated for individuals to understand.

This maze of details hides potential opportunities for simplifying and regularizing the conceptual structure of the system to be re-engineered, and makes it difficult to recognize deficiencies in design and architectural structure. The amplification process implicit in constructing skeletal prototypes helps expose such opportunities.

We found that there are fundamental conceptual errors embodied in the legacy structures and algorithms. Some of those errors were exposed when structural asymmetries and irregularities are discovered in the process of extracting a model of the legacy software. Others were discovered only with the help of the oversimplified models that are common in the early stages of prototyping a proposed new architecture. Constructing a small and simple instance of the proposed architecture raises many of the main design issues, and the simplicity of the model makes it much easier to consider and evaluate alternative designs to find improved structures.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. The UML interaction diagrams lack the preciseness to support automatic code generation for the executable prototype. This weakness can be remedied by the use of the prototype language PSDL [12, 13] and the CAPS prototyping environment, which provide effective means to model the system's dynamic behavior in a form that can be easily validated by user via prototype demonstration.

References

[1] A. Aho, "Pattern Matching in Strings", in *Formal Language Theory: Perspectives and Open Problems*, R. Book (editor), Academic Press, NY, 1980, pp. 325-347.

[2] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, *Re-engineering the Janus(A) Combat Simulation System*, Technical Report NPS-CS-99-004, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.

[3] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, "Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping," *Design Automation for Embedded Systems,* 5(3/4), August 2000, pp.251-263. A preliminary version of the paper also appeared in *Proceedings of the 10th IEEE International Workshop in Rapid Systems Prototyping*, Clearwater Beach, Florida, 16-18 June 1999, pp. 216-221.

[4] D. Berry, Formal Methods: The Very Idea, "Some Thoughts About Why They Work When They Work," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems,* 1998, pp. 9-18.

[5] O. Bray and M. Hess, "Reengineering a Configuration-Management System," *IEEE Software*, Vol. 12, No. 1, Jan. 1995, pp. 55-63.

[6] V. Cabaniss, B. Nguyen and J. Moregenthaler, "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems," *IEEE TSE*, Vol. 24, No. 7, July 1998, pp. 534-558.

[7] *Janus 3.X/UNIX Software Programmer's Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[8] *Janus 3.X/UNIX Software Design Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[9] *Janus Version 6 User's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[10] *Janus Version 6 Data Base Management Program Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[11] S. Jarzabek and P.K. Tan, "Design of a Generic Reverse Engineering Assistant Tool," *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, 1995, pp. 61-70.

[12] B. Kraemer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, May 1993, pp. 453-477.

[13] Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 14, No.10, October 1988, pp. 1409-1423.

[14] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, Vol. 5, No. 2, 1988, pp. 66-72.

[15] Luqi, "System Engineering and Computer-Aided Prototyping," *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, Vol. 6, No. 1, 1996, pp.15-17.

[16] Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo and B. Shultes, "The Story of Re-engineering of 350,000 Lines of FORTRAN Code," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, 23-26 October 1998, pp. 151-160.

[17] M. Moore and S. Rugaber, "Domain Analysis for Transformational Reuse," *Proceedings of 4th Workshop on Reverse Engineering*, IEEE Computer Society, 1997, pp. 156-163.

[18] J. Pimper and L. Dobbs, *Janus Algorithm Document, Version 4.0*, Lawrence Livermore National Laboratory, California, 1988.

[19] L. Rieger and G. Pearman, "Re-engineering Legacy Simulations for HLA-Compliance," *Proceedings of the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*, Orlando, Florida, December 1999.

[20] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.

[21] *TAE Plus C Programmer's Manual (Version 5.1)*. Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

[22] J. Williams and M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X*, master's thesis, Naval Postgraduate School, Dept. of Computer Science, Monterey, CA, March 1999.

Modeling Constraints as Methods in Object Oriented Data Model

Samiran Chattopadhyay[1], Chhanda Roy[2], Swapan Bhattacharya[3]

[1]Department of Computer Sc. & Engg, Jadavpur University, Calcutta – 700032, INDIA.
E-mail: chttpdhs@yahoo.com

[2] R.C.C Institute of Information Technology, South Canal Road, Calcutta – 700 015, INDIA

[3] Indian Institute of Information Technology - Calcutta, Block FC, Sector – 3, Salt Lake,
Kolkata – 700 091, INDIA. E-mail: bswapan@hotmail.com

Abstract

Object-oriented databases are becoming increasingly popular because of their capabilities to provide rich semantic constructs to model real world entities and their relations [3, 6]. Modeling of constraints in Object Oriented Data Model has been the focus of attention of many researchers in recent times [1, 5, 9, 10]. In this paper, we have attempted to model single and multi-attribute constraints as special methods in a class. We have also modeled class constraints as methods in a collection class associated with a user-defined class. The syntax and semantics of such modeling is extended to deal with constraints in single inheritance. Finally, we have demonstrated the application of our constraint model by exploring the possibility of developing a pre-processor that would add validity code in the methods defined by the user. The development of such a pre-processor is to be based on a language and platform (such as Java) capable of run-time type identification, reflection and introspection.

Key words: Object Oriented Database, Constraints, UML, Inheritance, Java Reflection

1. INTRODUCTION

Object-oriented databases are becoming increasingly popular because of their capabilities to provide rich semantic constructs to model real world entities and their relations. In the process, the notions of object, class, inheritance, relationships among classes and objects have been thoroughly treated in object oriented database systems. But, a complete treatise on constraints is still evolving in the context of Object Oriented Data Model. However, the imperative need for integrity maintenance in database systems is a long recognized fact. Constraints are restrictions on properties and relations of database objects that ensure the integrity of data according to both the system and the user. Constraints also ensure that subsequent updates on data will not violate these restrictions. In recent years, modeling integrity constraints in object oriented databases has become an active research topic [1, 5, 9, 10]. In this paper, we have attempted to explore a way to statically model constraints as specialized methods by re-defining the model element "Class" and extending the meta-model of Object Oriented programming. Our objective in this paper is to clearly bring out the syntax and semantics constraints in an object oriented framework involving only single inheritance and usual relationship among classes. We have tried to demonstrate that such a modeling approach would enable us to construct a pre-processor that could add code to validate integrity constraints in the user-developed methods.

For more than a decade, specification, design and implementation of "Object Oriented Model" of Database systems has been the focus of many research efforts [3, 6]. Object Oriented data model is a logical organization of real world objects or entities, constraints on them and relationship among these objects. A core object-oriented data model consists of the following basic components, namely, object and object identifier, attributes, methods, class, class hierarchy and inheritance. Every object has a state, characterized by the set of values for the attributes of the object and a behavior, defined by the set of methods which manipulate the state of the object. Each attribute of a class of objects has an access specifier (such as public, private, protected etc.) that limits the visibility of the attribute. The state and behavior encapsulated in an object are accessed or invoked from the external world only through explicit message passing.

Inheritance is deriving a new class (subclass) form one or more existing classes(super classes). The subclass inherits all attributes and methods of existing classes and may have additional attributes and methods. Exact semantics of inheritance is, however, dependent on a object oriented language and thus, not universal. A class "Y" is said to be a subclass of class "X" (equivalently, class "X" is said to be a super class of class "Y") if and only if every object of class "Y" is necessarily an object of class "X" (that is, Y ISA X). Objects of class "Y" then inherits the instance variables and methods of class "X". As a consequence, we can always use a "Y" object wherever an "X" object is permitted (that is, as an argument to various methods). This is the principle of substitutability, which helps us to take the advantage of code reusability. The ability to apply the same methods to different classes, or rather the ability to apply different methods with the same name to different classes (a class "X" method might need to be redefined for use with class "Y") is referred to as polymorphism. Thus, when a class "Y" inherits from a class "X" then the set of instance variables, and the set of methods of Y are a superset of the set of instance variables and the set of methods of "X". The subclass "Y" can override the implementation of an inherited method or instance variable by providing an alternative definition or implementation of the base class. In this paper, we will deal with single inheritance only.

A prototype implementation of an Object Oriented Database (OODB) for VLSI designs have been developed in [8]. Jasmine, a full-fledged implementation of OODB, has been reported in [4]. [7] deals with an another advanced object modeling environment. But these papers do not provide any insight to handle integrity constraints in object oriented databases.

There has also been a good amount of effort to identify, specify, design constraints in an object oriented database [1, 5, 9, 10]. In relational databases, Date [2] identified four categories of integrity rules, namely domain rules, attributes rules, relation rules and database rules. In the context of OODB, integrity rules may also be categorized in four groups - domain rules, attributes Rules, class rules and database rules [5]. Class Rules apply to the objects of a given class only, while database rules apply to objects form two or more distinct classes. It has been argued in [5], the domain rules and attribute rules are represented and maintained in an OODB by the class hierarchy automatically by the virtue of object-orientation. Therefore, only class rules and database rules need to be specified.

In [1], constraints have been modeled by means of exceptions in an object-oriented database. Ou deals with the specification of integrity constraints such as key, uniqueness using UML class diagrams [9, 10]. Ou's work proposes two ways to specify integrity constraints in a class. One is to use a property string to specify attribute constraints and the other is to add a compartment to specify class constraints. Our work differs from Ou in a major way. Class constraints (rules) pertain to integrity constraints that may be checked after examining all objects of a class [9]. A non-static method in a class is automatically provided with a self-reference to the object on which the method is called. Static methods can manipulate only non-static members of objects of a class. Methods for checking class constraints must have access to all objects of the class and thus, they can not be methods of the class itself. Class constraints can only be attached to a system defined class containing all objects of a particular class. To reduce the number of class constraints for reasons of performance, we also have single and multi-attribute constraints. While single attribute constraints are methods attached to an attribute, multi-attribute constraints are specified as special methods of a class. We have demonstrated the correctness of our model when applied in a class hierarchy involving single inheritance.

The layout of the paper is as follows. We discuss our approach to model integrity constraints (single attribute, multi-attribute and class rules) in Section 2. The modeling approach has been shown to be extended to inheritance in Section 3. We have presented a set of user's code and a possible augmentation of constraint validation code by the system by a pre-processor that understands the semantics of our model for specifying constraints in Section 4. Finally, we conclude in Section 5.

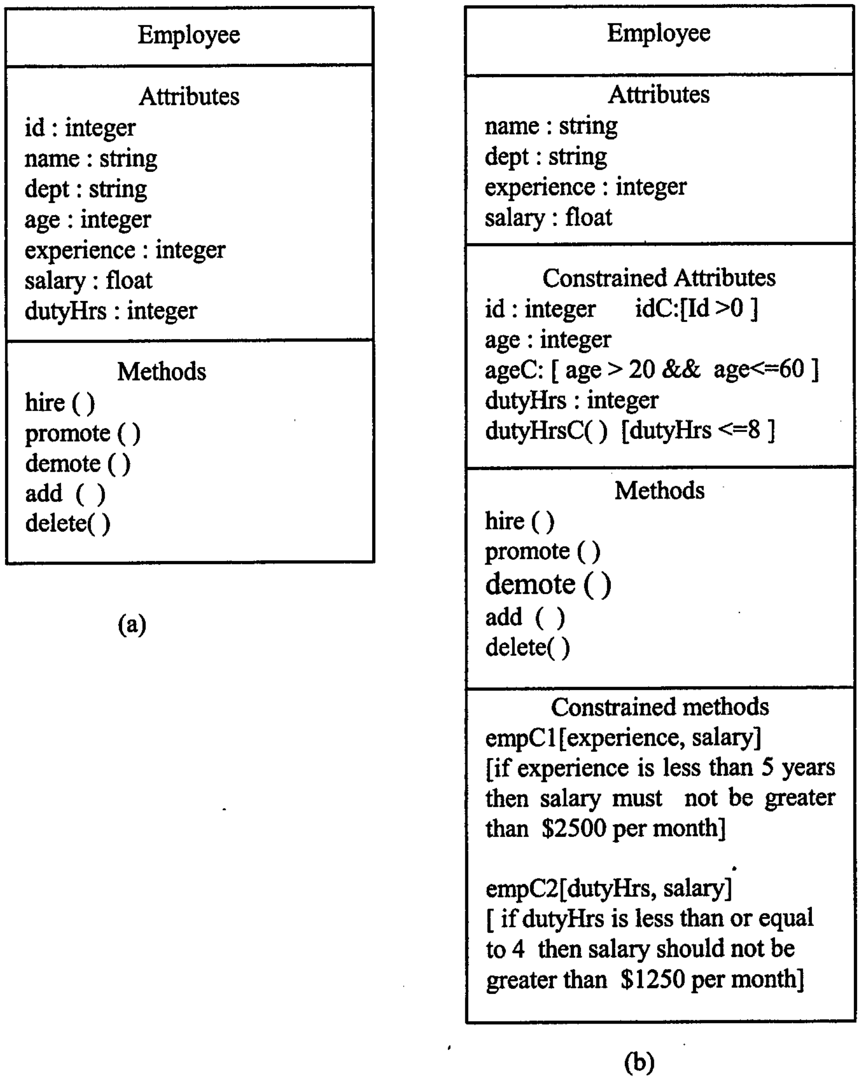
2. MODELING CONSTRAINTS

An integrity constraint is a semantic information in an object or a relationship among objects. A constraint specifies a condition and a proposition that must be maintained as true. Certain kinds of constraints are predefined; others may be user-defined. A user-defined constraint is described in words in a given language, whose syntax and interpretation is a tool responsibility. We attempt to model each constraint as a boolean method, which returns either true value or false value. If the predicate within a method is satisfied by a model element then the method will return true; otherwise the method returns false. These methods are clearly distinguished from the usual methods of a class by their usage and therefore, the constraint methods are accommodated differently than usual methods in object–oriented data model.

In this section, we will consider three different types of constraints present in an object oriented data model.

(i) Single Attribute Constraints,
(ii) Multiple Attributes Constraints &
(iii) Class Constraints.

Single attribute constraints are applicable to individual attributes of a class; multiple attributes constraints involve more than one attributes of a class and class constraints are applicable to individual classes of an object oriented database system. For example, suppose that there is a class "Employee" with usual attributes and methods as shown in Figure 1(a). A constraint on the attribute "age" of any "Employee" object may be described as "age of an employee must between 20 & 60". Similarly a constraint on "id" may be described as "id must be greater than 0".



(a)

(b)

Figure 1. (a) Class "Employee" (b) Class "Employee" with constraints

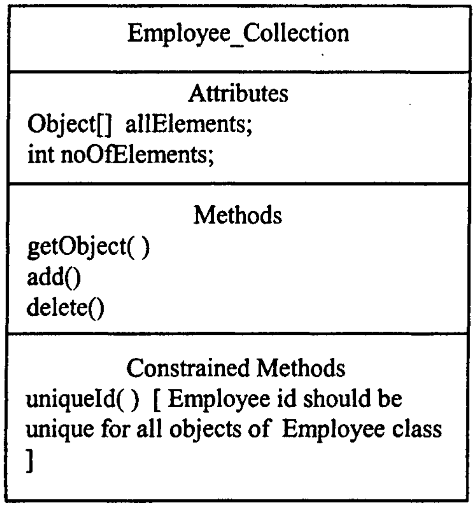
These constraints involve only one attribute of the class and they are called single attribute constraints. Similarly, there could be a constraint described as "if experience of an employee is less than 5 years then the salary of that employee can not be more than $2500". This constraint is an example of a multiple attribute constraint. A constraint, which can be checked by considering all objects of a class, is called a class constraint. A typical example of a class constraint in the "Employee" class may be described as "id of an employee must be unique". This constraint implies that whenever "id" field of an "Employee" object is modified, uniqueness of the updated "id" value must be guaranteed by checking "id" values of all other "Employee" objects.

The methods for the constraints idC and empC1 are shown in the following.

```
Boolean idC( )
        {
                if (empid <= 0)
                        return (false);
                return(true);
        }

Boolean empC1( )
        {
                if  ((salary > 2500) && (experience < 5))
                        return (false);
                return(true);
        }
```

In this section, we first deal with single and multi attribute constraints and then take up the class constraints. To handle single and multi-value constraints, we propose that the element "Class" be redefined as a 4-tuple <A, CA, M,CM> , where, as usual , "A" & "M" represent attributes and methods respectively of the class. And "CA" and "CM" represent constrained attributes and constrained methods respectively. A constrained attribute has the name of the attribute, its type and a boolean method representing a single attribute constraint. A constrained method represents a multi-attribute constraint and contains the names of the attributes involved.

Methods in a constrained attribute may be public, private or protected depending on the visibility of the individual attributes involved with the constraint methods. These constraint methods would be invoked for checking the validity of the attribute's value whenever the value of an object changes. If the attribute associated with a particular constraint method is private, then the visibility of that particular constraint method would also be private. This is because the value of a private attribute can only be changed by a member function or a friend function and a private constraint method can be invoked from within members or friends without causing any compilation or runtime error. Similarly, if the attribute associated with a particular constraint method is public or protected, then the visibility of the constraint method would also be public or protected respectively.



Employee_Collection

Attributes
Object[] allElements;
int noOfElements;

Methods
getObject( )
add()
delete()

Constrained Methods
uniqueId( ) [Employee id should be unique for all objects of Employee class]

Figure 2. The class "Employee_Collection"

To represent class constraints, we introduce a singleton collection class associated with each general user-defined class, where the singleton collection class would always contain a collection of all objects of the user-defined class. All constraints that need to check all objects of a user-defined class for validation become boolean methods of the singleton class. Thus, constraints on objects of the user's class can be validated by calling a corresponding method of the collection class. For example, suppose that there is a constraint on the class "Employee" which states that "id" of every "Employee" object must be unique. For the class "Employee", a collection class "Employee-Collection" is defined (see Figure 2). The class “Employee-Collection” has two attributes, a pointer to the list of all existing “Employee” objects and the number of “Employee” objects. The collection class also contains a method “uniqueId” to check the class constraint mentioned for the class “Employee”.

Similarly the singleton class which is the collection of all objects of the “Employee” class can be described as shown in Figure 2.

Whenever a new object of the class “Employee” is created, a reference to that object is added to the list “allElements” and the integer “noOfElements” is incremented. Similarly, whenever, an “Employee” object is deleted then the reference of that object is deleted from the list “allElements” and the integer “noOfElements” is decremented. Whenever the “id” of an object is modified by a statement in the user’s program, the system must call the class constraint method “uniqueId( )” of the collection class to check whether the “id” remains unique for all object of the “Employee” class even after the modification.

For linking the only instance of the class “Employee_Collection “ with the class an “Employee” object, the system first calls the class level method getobject( ) of the collection class. After getting the object of the singleton class it would call the constraint method of “Employee_Collection” class on that single object.

3. MODELING CONSTRAINT IN INHERITANCE

Let us assume that the class “Manager” is derived from the “Employee” class. We can add new constraints into the derived classes as we can add new attributes and new methods. Further, constraints specified in the base class “Employee” may be modified in the derived classes. Constrained attributes and constrained methods are inherited in the derived class. An (not constrained) attribute in the base class can be re-declared as constrained in the derived class. The methods in the constrained attributes may also be re-defined. New constrained attribute may be added. Constraint methods specified in base classes may be modified in derived classes. New constraint methods may also be added. Let us consider the class hierarchy as shown in Figure 3.

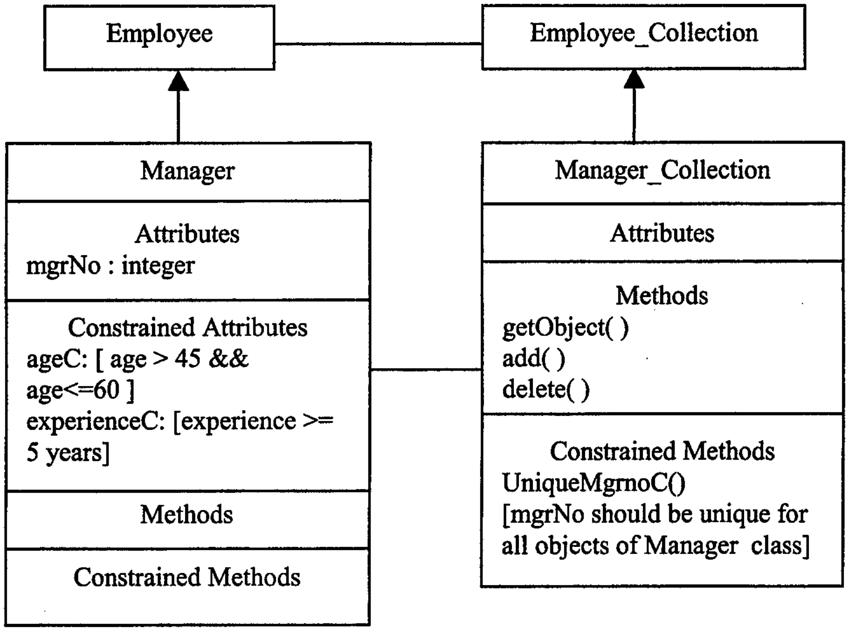


Figure 3. Class Hierarchy in modeling constraints in inheritance

In the given class hierarchy, we can add a new single attribute constraint method “experienceC( )” in the “Manager” class as “Experience must be greater than or equal to five years”. The base class attribute constrained method “ageC( )” [“Employee age should be greater than 20 years and less than or equal to 60 years”] may be modified in the derived “Manager” class such as “ Manager’s age should be greater than 45 years”. Allowing modification to constraints in a derived class which are already specified in the base class poses a difficult logical problem. It is possible that the re-definition leads to an inconsistency in the base class. For example, if the method “ageC( )” is modified as “Manager’s age should be greater than 45 years and less than 65 years", then instances of "Manager" class would not remain valid the instances of "Employee" class. Although this seems to be a serious restriction, if the reference can always be resolved to the run-time objects, then the problem may implementionally be overcome. For the time being, we leave it to the database designers to take care of the issue of consistency.

We can further add new class constraints in "Manager" class such as "Manager number should be unique", "Department code for each Manager should be unique" etc. These class constraint methods should be added into the singleton class "Manager_Collection" class, which contains a collection of all objects of the "Manager" class. If a user modifies "mgrNo" attribute of a manager object "ml", the system adds a call to the corresponding class constraint methods of "Manager_Collection" class. The class "Manager_Collection" is to be derived from "Employee_Collection " so that class constraints in the base class are inherited in the derived class.

Defining new multiple attribute constraints and class constraints in a derived class poses some problems when the new constraints involve attributes in the base class. In such cases, all base class methods that modifies the values of one attribute involve in the new constraint must be re-written (re-defined) in the derived class. The re-defined method should check the validity of the modified value of the said attribute by invoking appropriate constraint method.

4. APPLICATION OF CONSTRAINT MODEL

In this section, we explore the possibility of using the semantics of our constraint model to construct a preprocessor that would add constraint validation code to user's programs. We demonstrate that developing such a preprocessor is indeed possible by looking at the following example scenarios. The classes that we are considering are the class "Employee" and the class "Manager" derived from "Employee". We also have two collection classes, "Manager_Collection" derived from "Employee_Collection". For simplicity, we make the following assumptions.

- The attributes of any object are not supposed to be directly updated even by a method in the class itself. That is, a method has to be invoked to set attributes of an object.
- An instance of a class is to be created by calling a "factory" method such as "create" present in every class.
- We further assume that all attributes of the classes "Employee" and "Manager" are constrained. That is, for an attribute "A", without constraint, we assume that there is an empty function "AC" representing the constraint for "A" making the attribute "A", a constrained attribute.
- We assume that the preprocessor is developed on a language such as "Java" which has strong run-time type identification, reflection and introspection capabilities. Java allows us to perform the following operations at run-time:
  (i) find out the methods, attributes, super classes of a class whose name is known at run-time;
  (ii) create an object of a class which is known at run time.
  (iii) accessing modifying the properties of an object where the property names are known at run-time;
  (iv) invoking methods on an object where the method names are known at run-time;
- The singleton objects of the collection classes "Employee_Collection" and "Manager_Collection" are pre-created with appropriate initializations of the array "allElements" present in these objects. That is, the "allElements" array of these objects contains references to existing "Employee" and "Manger" objects.

Next, we take up sample scenarios to explain the working of the preprocessor.

Scenario1: Creation of an object through factory methods in a class.

Suppose that the factory method of the class "Employee" dynamically creates a new "Empolyee" object.

```
Employee Create( )
{
            Employee e = new Employee();
            return(e);
}
```

The preprocessor modifies the method "Create" in the following manner.

```
Employee Create()
{
        Employee e = new Employee();
        Class c = get the “Class’ object of  the class whose name is “Employee_Collection”;
        Object o = invoke the method “getObject” of the Class “C”;
        Invoke “add” method by passing (Object)e on the object “o”;
        c = super class of c;
        while(c is not null)
        {
                o = invoke the method “getObject” of the Class “C”;
                Invoke “add” method by passing (Object)e on the object “o”;
        }
}
```

Note that whenever an object is added to a collection class, “X”, the same object is also added to all super classes of “X”.

Scenario 2: Updation of an attribute which has an attribute-constraint.

Suppose that “setAge” is a function in the “Employee” class as described in the following.

```
        void setAge(int x)
        {
                age = x;
        }
```

The code generated by the preprocessor for the above function is shown in the following.

```
        void setAge(int x)
        {
                age = x;
                if (ageC() = = false)
                        goto errorhandler;
        }
```

In case “ageC” is re-defined in the class “Manager’ and the object on which “setAge” is called be a “Manager” object then automatically “ageC” function as defined in “Manager’ class would be called.

Scenario 3. Updation of an attribute which has a class constraint.

Suppose that “setId” is a function in the “Employee” class which is as follows.

```
void setId(int x)
        {
                id = x;
        }
```

The modified code of the function “setId” as produced by the preprocessor would be as follows.

```
        void setId(int x)
        {
                id = x;
                n= get the run-time class name of “this” object;
                n1= construct the class name of the corresponding collection_class;
                o= invoke “getObject” method of the class where name is “n1”;
                invoke the “uniqueId” method of the object “O”;
        }
```

Even if "setId" is invoked on a variable "e" which actually refers to a "Manager" object, the preprocessor generated version of "setId" would invoke the "uniqueId" method of the appropriate class at execution time.

5. CONCLUSION

Modeling of constraints in OODM has been the focus of attention of many researchers in recent times. In this paper, we have attempted to model single and multi-attribute constraints as special methods in a class. We have also modeled class constraints as methods in a collection class associated with a user-defined class. The syntax and semantics of such modeling is extended to deal with constraints in single inheritance. Finally, we have demonstrated the application of our constraint model by exploring the possibility of developing a pre-processor that would add validity code in the methods defined by the user. The development of such a pre-processor is to be based on a language and platform capable of run-time type identification and introspection.

Modeling constraints as methods have a serious logical problem as methods in the base class may be re-defined in the derived class leading to inconsistent situations. We have assumed that it is the responsibility of the database designer to ensure that inconsistency is avoided in the overall definition of the system. We have not considered the issue of multiple inheritance either. It may be interesting to see how constraints in multiple inheritance may be modeled. Similarly, modeling database constraints (like foreign key etc.) would also be challenging task. We are in the process of implementing the pre-processor we discussed in our paper, the prototype of which would be ready soon.

References

[1] Bassiliades, N., and Vlahavas, I., "Modeling constraints with exceptions in object oriented Database", Proceedings of 13th International Conference on the Entity-Relationship Approach, Manchester, United Kingdom, December 1994, Lecture Notes in Computer Science, P. Loucopoulos (Ed.), Vol. 881, 189-201, 1994.

[2] Date, C.J., "An Introduction to Database Systems", Sixth Edition, Addision- Wesley Publishing Company Inc., 1995.

[3] Gray, P.M.D., Kulkarni, K. G., Paton, N.W., "Object-Oriented Databases – A semantic Data Model Approach", Prentice Hall International, 1992.

[4] Ishikawa, H., Yamane, Y., Izumida Y., & Kawato, N., "An object-oriented Database System Jasmine : Implementation, Application and Extension" , IEEE Transaction on Knowledge and data Engineering ,Vol 8, No 2, April 1996.

[5] Jagadish, H.V., Qian, X., "Integrity Maintainance in an Object Oriented Database", Proc. Of the 18th International Conference on Very Large Databases, (Vancouver, BC, Canada, August, 1992).

[6] Khoshafian, S., "Object –Oriented Databases", John Wiley & Sons, Inc, 1993.

[7] Li, Q., Lochovsky, F. H., "ADOME : An advanced object Modeling Environment ", IEEE Transaction on Knowledge and data Engineering ,Vol 10, No 2, March/April 1998.

[8] Nayak, T. K., Majumdar, A.K., Basu, A. & Sarkar, S., "VLODS : A VLSI OBJECT ORIENTED DATABASE SYSTEM ", Information Systems Vol 16, No 1, 73-96, 1991.

[9] Ou, Y., "On using UML class Diagram for object oriented database Design - Specification of integrity constraints", International Workshop, UML'98, (Mulhouse, June 1998).

[10] Ou, Y., "On mapping between UML and Entity Relationship Model", International Workshop, UML'98, (Mulhouse, June 1998).

A Unified Approach for the Integration of Distributed Heterogeneous Software Components[1]

Rajeev R. Raje[2] [3]   Mikhail Auguston[4] [5]   Barrett R. Bryant[4] [6]   Andrew M. Olson[2]   Carol Burt[7]

Abstract

Distributed systems are omnipresent these days. Creating efficient and robust software for such systems is a highly complex task. One possible approach to developing distributed software is based on the integration of heterogeneous software components that are scattered across many machines. In this paper, a comprehensive framework that will allow a seamless integration of distributed heterogeneous software components is proposed. This framework involves: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glues and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of quality of service (QoS) offered by each component and an ensemble of components. A case study from the domain of distributed information filtering is described in the context of this framework.

Keywords: Distributed systems, Formal methods, Glue and Wrapper technology, Quality of Service

1 Introduction

The rapid advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. This change in paradigm is allowing us to develop distributed computing systems (DCS). DCS appear in many critical domains and are, typically, characterized by: a) a large number of geographically dispersed and interconnected machines, each containing a subset of the required data, b) an open architecture, c) a local autonomy over the hardware and software resources, d) a dynamic system configuration and integration, e) a time-sensitivity of the expected solution, and f) the quality of service with an appropriate notion of compensation. These characteristics make the software design of DCS an extremely difficult task.

One promising approach to the software design of DCS is based on the principles of distributed component computing. Under this paradigm DCS are created by integrating geographically scattered heterogeneous software components. These components constantly discover one another, offer/utilize services, and negotiate the cost and the quality of the services. Such a view provides a scalable solution and hides the underlying heterogeneity.

Various distributed component models, each with strengths and weaknesses, are prevalent and widely used. However, almost a majority of these models have been designed for 'closed' systems, i.e., systems, although distributed in nature, are developed and deployed in a confined setup. In contrast, a direct consequence of the heterogeneity, local autonomy and the open architecture is that the software realization of DCS requires combining components that adhere to different distributed models. This in turn increases the complexity of the design process of DCS. Hence, a comprehensive framework, that provides a seamless access to underlying components and aids in the design of DCS, is needed.

In this paper, one such framework is described. This framework consists of: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glue and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of the notion of quality of service offered by each component and an ensemble of components. The paper also presents a case study that shows the application of the framework to a specific problem domain.

The rest of the paper is organized as follows. The next section contains a detailed discussion about the meta-model. As an application of the meta model, a case study from the domain of distributed information filtering is presented in the Section 3. Section 4 deals with the formal specification of the meta model, the automated system integration, and evaluation of the approach. Finally, we conclude in Section 5.

[1] This material is based upon work supported by, or in part by, the U. S. Office of Naval Research under award number N00014-01-1-0746.

[2] Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202, USA, {rraje, aolson}@cs.iupui.edu, +1 317 274 5174/9733

[3] This material is based upon work supported by, or in part by, the National Science Foundation Digital Libraries Phase II grant.

[4] Computer Science Department, Naval Postgraduate School, 833 Dyer Rd., SP 517, Monterey, CA 93943, USA, {auguston, bryant}@cs.nps.navy.mil, +1 831 656 2509/2726

[5] This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number 40473-MA. On leave from Computer Science Department, New Mexico State University, USA.

[6] This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350. On leave from Department of Computer and Information Sciences, University at Alabama at Birmingham, USA.

[7] 2AB, Inc., 1700 Highway 31, Calera, AL 35040, USA, cburt@2ab.com, +1 205 621 7455

2 Component Models and a Meta-model

Many models and projects for the software realization of DCS have been proposed by academia and industry. A few prominent ones are: Java^{TM} Remote Method Invocation (RMI) [16], Common Object Request Broker Architecture (CORBA^{TM}) [16, 20], Distributed Component Object Model (DCOM^{TM}) [11, 16], Web-component model/DOM [10], Pragmatic component web [5], Hadas [6], Infospheres [4], Legion [22], and Globus [21]. Each of these models/projects has strength and weaknesses. Some of these are language-centric and only assume a uniform way of the world (Java); while the others allow a limited interoperability (CORBA - allowing implementations in different languages). Some of these are general-purpose, i.e., not concentrating on any particular application domain (DCOM), while others are specifically tailored to high-performance computing applications (Legion). However, almost all of these models/projects do not assume the presence of other models. Thus, the interoperability which they provide is limited mainly to the underlying hardware platform, operating system and/or implementational languages. Also, there are hardly any models which emphasize the notion of quality of service offered by the components. Projects, such as Agent TCL [8], etc., based on the principles of intelligent agents have imbibed the notion of the quality of service and related compensation. However, the agents are at a higher level of abstraction than components and many of the agent projects/frameworks use one or the other existing distributed-component models at the low-level.

2.1 Why a Meta-model?

Given the above mentioned plethora of component-based models and also noting the fact that components, by their definition, are independent of the implementation language, tools and the execution environment; it is necessary to answer the questions: *why is a meta-model needed for a seamless interoperation of distributed heterogeneous components?* and *how would a meta-model assist in seamlessly integrating distributed heterogeneous software components?* The answer to these question lies in: a) in any organization, software systems undergo changes and evolutions, b) local autonomy is an inherent characteristic of today's geographically (or logically) dispersed organizations, and c) if reliable software needs to be created for a DCS by combining components then the quality of service offered by each component needs to become a central theme of the software development approach.

The consequence of constant evolutions and changes is that there is a need to rapidly create prototypes and experiment with them in an iterative manner. Thus, there is no alternative but to adhere to cyclic (manual or semi-automatic) component-based software development for DCS. However, the solution of decreeing a common COTS environment, in an organization, is against the principle of local autonomy. Hence, the development of a DCS in an organization will, most certainly, require creating an ensemble of heterogeneous components, each adhering to some model. Also, every DCS is designed and developed with a certain goal in mind, and usually that goal is associated with a certain perception of the quality (as expected from the system) and related constraints.

Thus, there is a need for a comprehensive meta-model that will seamlessly encompass existing (and future) heterogeneous components by capturing their necessary aspects, including the quality of service offered by each component and an amalgamation of components.

2.2 Unified Meta-component Model (UMM)

In [17] we have proposed a unified meta-component model (UMM) for global-scale systems. The core parts of the UMM are: *components*, *service and service guarantees*, and *infrastructure*. The innovative aspects of the UMM are in the structure of these parts and their inter-relations. UMM provides an opportunity to bridge gaps that currently exist in the standards arena. For example, the CORBA Component Model (CCM^{TM}) [13] and Java Enterprise Edition component models (J2EE^{TM}) are consistent, and yet, because of the absence of a formal meta-model, it is difficult during the evolution of each to recognize when the boundaries that maintain the consistency are crossed. Similarly, it has been demonstrated in numerous products that the Component Object Model (COM^{TM}) [18] and CORBA component models are similar (in an abstract sense) enough to allow meaningful bridging. It is, however, not possible to point to a Meta-model that constrains the implementations of these technologies.

For enterprise component solutions, this is an area where significant standards work is now focused. The OMG Meta Object Facility (MOF^{TM}) [14] provides a common meta-model that allows the interchange of models between tools as well as the expression of models in XMI^{TM} (an MOF compliant XML^{TM} (eXtended Markup Language)) [12]. This work allows the generation of interfaces from Unified Modeling Language (UML) [19] models, however, a careful analysis of the resulting interface specifications makes it clear that distribution is not a key factor in the algorithms used. For example, quality of service requirements for performance, scalability and/or security would dictate the use of iterators, the factoring of interfaces to separate "query" and "administrative" operations, and the use of structures and/or objects passed by value. The current standards in this tend to focus on data access with accessors and mutators and relationship transversal. This is acceptable in a single machine environment, but unacceptable for highly distributed communications and collaborations. The recent shift in focus for the Object Management Group to "Model Driven Architecture" (MDA^{TM}) [15] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization of Business and Component Meta-Models. The need to support the evolution of component models and to describe the capabilities of the models will be key to realizing the full potential of an E-business economy.

The following sections describe the various aspects of UMM in detail.

2.2.1 Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-component model and there is no notion of either a centralized controller or a unified implementational framework. Each component has a state, an identity and a behavior. Thus, all components have well-defined interfaces and private implementations. In addition, each component in UMM has three aspects: 1) a computational aspect, 2) a cooperative aspect, and 3) an auxiliary aspect.

Computational Aspect

The computational aspect reflects the task(s) carried out by each component. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. In DCS, components must be able to 'understand' the functionality of other components. Thus, each component in UMM supports the concept of introspection, by which it will precisely describe its service to other inquiring components. There are various alternatives for a component to indicate its computation – ranging from simple text to formal descriptions. Both these extremes have advantages and drawbacks. UMM takes a mixed approach to indicate the computational aspect of a component – a simple textual part, called *inherent attributes* and a formal precise part, called *functional attributes.*

The functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service offered by the component. Multi-level contracts for components have been proposed by [2], classifying the contracts into four levels – syntactic, behavioral, concurrency and quality of service (QoS). UMM integrates this multi-level contract concept into the functional part of the computational aspect. As stated earlier, in DCS each component will be offering a service and hence, the level related to the QoS is especially critical in UMM. The QoS depends upon many factors such as, the algorithm used, the execution model, resources required, time, precision and classes of the results obtained. UMM makes an attempt at quantifying the QoS by creating a vocabulary and providing multiple levels of quality, which could be negotiated by the components involved in an interaction. The functional part will also be specified by the creator of the component.

Cooperative Aspect

In UMM, components are always in the process of cooperating with each other. This cooperation may be task-based or greed-based. The cooperative aspect depends on many factors: detection of other components, cost of service, inter-component negotiations, aggregations, duration, mode, and quality. Informally, the cooperative aspect of a component may contain: 1) Expected collaborators – other components that can potentially cooperate with this component, 2) Pre-processing collaborators – other components on which this component depends upon, and 3) Post-processing collaborators – other components that may depend on this component.

Auxiliary Aspect

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of DCS. The auxiliary aspect of a component will address these features. In UMM, each component can be potentially mobile. The mobility of the component will be shown as a 'mobility attribute' (a notion similar to the inherent attribute). If a component is mobile, then the mobility attribute will contain the necessary information, such as its implementation details and required execution environment. Similarly, security in DCS is a critical issue. The security attribute of a component will contain the necessary information about its security features. As DCS are prone to frequent failures, full and partial, fault tolerance is critical in these systems. Similar to mobility and security, each component contains fault-tolerant attributes in its auxiliary aspect.

2.2.2 Service and Service Guarantees

The concept of a service is the second part of the UMM. A service could be an intensive computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify the cost and quality of the service offered.

The nature of the service offered by each component is dependent upon the computation performed by that component. In addition to the algorithm used, expected computational effort and resources required, the cost of each service will be decided by the motivation of the owner and the dynamics of supply and demand. In a dynamic environment costs must always be accompanied by the duration for which the costs are valid. As the system dynamics undergo constant changes, the methodologies used to fix the cost of a service will evolve as time progresses, thereby creating a need to indicate the time sensitiveness of the cost. The quality of service is an indication given by an component, on behalf of its owner, about its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures. The techniques used to determine the cost, the time-validity and the quality of a service will depend upon the tasks carried out by the component and the objectives of its owner and will involve principles of distributed decision making.

There are many parameters that a component can use to indicate its quality of service. A few examples are: i) Throughput – number of methods executed per second and classification of methods based on their read/write behaviors, ii) Parallelism constraints – synchronous or asynchronous, iii) Priority, iv) Latency or End-to-End Delay – turn-around time for an invocation, v) Capacity – how many concurrent requests a given component can handle, vi) Availability – indication of the reliability of a component, vii) Ordering constraints – can invocations (asynchronous) be executed out of order by a component, viii) Quality of the result returned – does the component provide a classification or ranking of the result, and ix) Resources available – how many resources (hardware/data) are accessible to the component under consideration and what are the types of resources.

When a component uses certain metrics to indicate its QoS (either all the mentioned criteria or a sub/super set of them), three interesting issues need to be addressed: a) how does the component developer decide these parameters?, b) how does the developer guarantee the advertised QoS during the execution?, and c) when components are collected together as a solution for specific DCS, what happens to the QoS of the combination and how does the combined QoS meet the quality requirements of DCS?

The parameters to be used to describe the QoS of a component are highly context (application) dependent. The proposed approach is to create lists of QoS metrics for common application domains. A few examples of such domains are: scientific computing, multi-media applications, information filtering, and databases. Once such lists are created, they would be used as a template by the component developers while advertising the QoS of their components.

QoS of Components

The issue of guaranteeing a particular QoS, for a component, in an ever changing dynamic DCS is extremely critical; mainly because of external (e.g., policy matters related to resources) and internal (e.g., changes in algorithms) factors that affect a life cycle of a component. In addition, as the software realization of DCS is based on an amalgamation of heterogeneous components, a proper guarantee of a QoS offered by a component effectively decides the QoS of the entire DCS. The quality metrics are expected to vary from one application domain to another and which metrics to select would depend on the intentions of the component developer and the functionality offered by that component. A few examples of such QoS metrics are already mentioned in the previous section. Irrespective of the metrics selected, there is a need for a well-defined mechanism that will assist the developer to achieve the necessary QoS when that component is deployed. Just like any software development process, the process of guaranteeing a certain QoS, as offered by a component, will be an incremental and iterative one, as will be discussed later.

QoS of an Integrated System

In addition to the QoS of individual components, there is a need to achieve a certain QoS for the ensemble of heterogeneous components assembled for a distributed system under discussion. The QoS of such an amalgamation will be decided by the design constraints of the system under construction. However, the integral characteristics of such a system typically cannot be expressed as a function of individual components but as a property of the whole system behavior. Hence, there is a need for a formal model of system behavior, which will integrate the behaviors of each component in the ensemble along with its QoS guarantees.

The proposed approach to address the problem of QoS is as follows. First, build a precise model of systems behavior (event trace notion), provide a programming formalism to describe computations over event traces, and then apply these in order to define different kinds of QoS metrics. Constructive calculations of QoS metrics on a representative set of test cases is one of cornerstones of the proposed iterative approach to system assembly from components meeting user's query specifications.

This approach to the design of a system behavior model assumes that the run time actions performed within the system may be observed as detectable events. Each event corresponding to an action is a time interval, with beginning, end, and duration. Certain attributes could be associated with the event, e.g. program state, source code fragment, time, etc. There are two binary relations defined for the event space: inclusion (one event may be nested within another), and precedence (events may be partially ordered accordingly to the semantics of the system under consideration). Hence, when executed, a system generates an event trace - set of events structured along the relations above. This event trace actually can be considered as a formal behavior model of the system ("lightweight semantics"). This model could be presented as a set of axioms about event trace structure called event grammar [1].

For example, suppose that the entire system execution is represented by an event of type execute-system. It may contain events of the type evaluate-component-A and evaluate-component-B. Event grammar may contain an axiom:

```
execute-system:  (evaluate-component-A evaluate-component-B)*
```

which states that `evaluate-component-A` is always followed by the `evaluate-component-B` event, and these pairs may be repeated zero or more times.

A new concept for specification and validation of target program behavior based on the ideas of event grammars and computations over program execution traces has been developed, and assertion language mechanisms, including event patterns and aggregate operations over event traces, to specify expected behavior, to describe typical bugs, and to evaluate debugging queries to search for failures (e.g. gathering run time statistics, histories of program variables, etc.) have been created. An event grammar provides a basis for QoS metrics implementation via target program automatic instrumentation. Since the instrumentation is conditional, it does not deteriorate the efficiency of the final version generated code. This mechanism based on independent models of system behavior makes it possible to define QoS metrics as generic trace computations, so that the same metric may be applied to different versions of an assembled system (via automatic instrumentation). To facilitate use of the event grammar model for the assembled system, the event definitions should be consistent through the entire component space. The QoS metrics for components should adhere to this principle. The process proposed in Section 4.4 for assembling a distributed system from components in a distributed network offers a possible approach to achieving this.

2.2.3 Infrastructure

As local autonomy is inherent in open DCS, forcing every component developer to abide by certain rigid rules, although attractive, is doomed to fail. UMM tackles the issue of non-uniformity with the assistance of the *head-hunter* and *Internet Component Broker.* These are responsible for allowing a seamless integration of different component models and sustaining a cooperation among heterogeneous (adhering to different models) components.

Head-hunter Components

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt at match-making between service producers and consumers. A head-hunter is analogous to a binder or a trader in other models, with one difference – a trader is passive, i.e., the onus of registration is on the foreign components and not on the trader. In contrast, a headhunter is active, i.e., it discovers other components and makes an attempt to register them with itself. There are many approaches possible for the discovery of components. They range from the standard search techniques to broadcasts and multi-casts to selected machines. At a conceptual basis, UMM does not tie itself to a specific approach but during the prototype development a particular approach will be selected for the discovery process. During registration, each component will inform the head hunter about all its aspects. The head hunter will use this information during matching. A component may be registered with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components. The functionality of head hunters makes it necessary for them to communicate with components belonging to any model, implying that the cooperative aspect of head hunters be universal. Considering the heterogeneous nature of the components, it is conceivable that the software realization of a distributed system will require an ensemble of components adhering to different models. This requires a mediator, the *Internet Component Broker*, that will facilitate cooperation between heterogeneous components.

Internet Component Broker

The Internet Component Broker (ICB) acts as a mediator between two components adhering to different component models. The broker will utilize adapter technology, each adapter component providing translation capabilities for specific component architectures. Thus, a computational aspect of the adapter component will indicate the models for which it provides interoperability. It is expected that brokers will be pervasive in an Internet environment thus providing a seamless integration of disparate components. Adapter components will register with the ICB and while doing so they will indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head hunter component will not only search for a provider, but it will also supply the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [9]. A reliable, flexible and cost-effective development of wrap and glue is realized by the automatic generation of glue and wrappers based on component specifications. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ORB provides the capability to generate the glue and wrappers necessary for objects written in different programming languages to communicate transparently; the ICB provides the capability to generate the glue and wrappers necessary for components implemented in diverse component models (and providing service guarantees) to collaborate across the Internet. An ORB defines language mappings and object adapters. An ICB must provide component mappings and component model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), the ICB will provide key features that are unique; it is designed to provide the auxiliary aspects of the Internet – collaboration between autonomous environments, mobility and security. In addition, the UMM includes quality of service and service guarantees. The ICB, in conjunction with head-hunters provide the infrastructure necessary for scalable, reliable, and secure collaborative business using the Internet.

3 A Case Study

In order to explain the UMM and the proposed approach, below a case study from the domain of distributed information filtering is presented. Although the case study uses a specific domain, the principles can be easily extended to other application domains that involve the software realization of a DCS.

3.1 Distributed Information Filtering

It is desired to develop a global information filtering system, in which, users will be interested in receiving selected information, based on their preferences, from scattered repositories. Usually, a filtering task involves contacting the scattered resources, performing an initial search to gather a subset of documents, representing, classifying and presenting based on the user profile. Many different methods are employed for the sub-tasks involved in filtering. Thus, it can be easily envisioned that different components, each employing a different algorithm to perform these sub-tasks, will be scattered across an interconnected system. Each component may belong to a different model, may quote different costs and offer different qualities of service.

Hence, a typical distributed information filtering system consists of the following types of components: a) Domain Component (DC), b) Wrapper Component (WC), c) Representer Component (RC), d) Classifier Component (CC), and e) User Interaction Component (UIC). In addition to these domain-specific components, headhunter components (HC) and the ICB are needed.

All these components, their aspects and characteristics need to be defined using UMM. For the sake of brevity, only the complete description of the domain component (DC) is shown below.

3.2 Domain Component

The domain component is responsible for maintaining a repository of URLs of associated information sources for particular type (e.g., text, structure, sequence) of information that needs filtering.

For example, the inherent attributes might consist of Author (name of the component developer), Version (current version of the component), Date Deployed, Execution Environment Needed and Component Model (e.g., Java-RMI 1.2.2), Validity (e.g., one month from the deployment), Atomic or Complex (indivisible or an amalgamation of other components, e.g. atomic), Registrations (with which headhunters this component is registered, e.g., H1 – `www.cs.iupui.edu/h1` and H2 – `www.cis.uab.edu/h2`).

An informal description of the functional part of a component may contain:

```
1. Computational Task Description -- e.g., searching a selected set of databases over the Internet.
2. Algorithm Used and its Complexity -- Webcrawling and O(n^2), respectively.
3. Alternative Algorithms -- Indexing.
4. Expected Resources (best, average and worst-cases) -- multi-processor, uni-processor (300MHz
with an CPU utilization of 50%), and uni-processor (100MHz with CPU utilization of 99%), respectively.
5. Design Patterns Used (if any) -- Broker.
6. Known Usages  -- for assembling an up-to-date listing containing addresses of known information
repositories for a particular domain.
7. Aliases--  such a component is usually called a Pro-active Agent.
8. Multi-level contracts:
e.g., for a function like List getURLs (Domain inputDomain, Compensation inputCost), the behavioral
contract could specify the pre-condition to be (valid Domain Name and cost), post-condition to be:
if successful (activeClientThreads++ and cost+=inputCost)
else (raise DomainNotKnownException and InvalidCostException)
and the invariant could be (ListOfURLs > 1).  Also, for the same function, the concurrency contract
could specify (maximum number of active threads allowed = 50).
```

The cooperation attributes of the domain component may consist of 1) expected collaborators UIC, WC, HC, TC and RC, 2) pre-processing collaborators HC and TC, and 3) post-processing collaborators RC and UIC.

The auxiliary attributes of the domain component are 1) fault-tolerant attributes, e.g., check-pointing versions, 2) security attributes, e.g., simple encryption, and 3) mobility attributes, e.g.. "not mobile."

For the domain component, the QoS parameters may contain 1) number of available URL's, 2) ranking of URL's, and 3) average rate of URL collection.

A component developer may offer several possible levels of QoS, e.g., L1) novice (number of URL's < 50 and no ranking of URL's and average rate of URL collection ≥ 1 week and average latency ≥ 2 minutes), L2) intermediate (number of URL's < 500 and simple ranking of URL's and average rate of URL collection ≥ 3 days and average latency ≥ 1 minute), and L3) expert (number of URL's < 1500 and advanced ranking of URL's and average rate of URL collection ≥ 1 day and average latency ≥ 5 seconds).

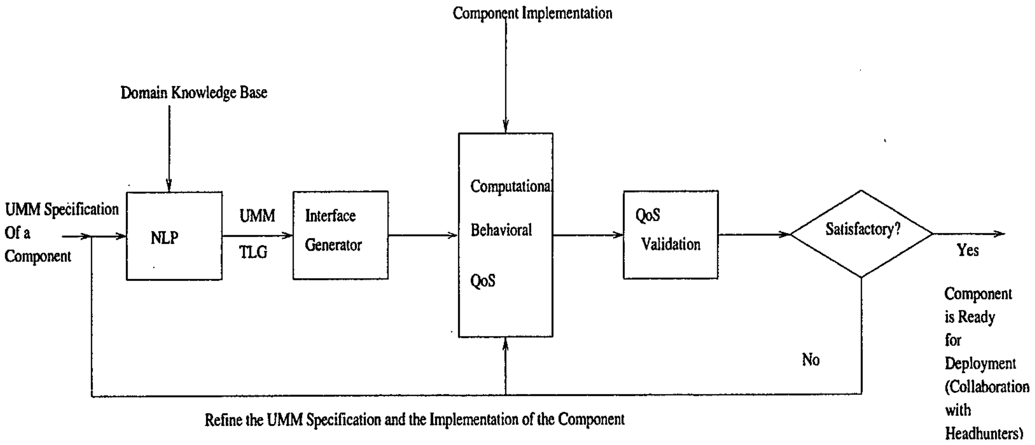


Figure 1: The Component Development and Deployment Process in UMM

The expected compensations for the above levels in terms of the number of URLs could be 1) L1 > 100 and < 200, 2) L2 > 200 and < 400, and 3) L3 > 400 and < 600.

4 Component and System Generation Using UMM Framework

The development of a software solution, using the UMM approach, for a DCS has two levels: a) component level – in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network, and b) system level – this level concentrates on assembling a collection of components, each with a specific functionality and QoS, and semi-automatically generates the software solution for the particular DCS under consideration. These two levels and associated processes are described below.

4.1 Component Development and Deployment Process

The component development and deployment process is depicted in Figure 1. As seen in the figure, this process starts with a UMM specification of a component (from a particular domain). This specification is in a natural-language format, as illustrated in the previous section. This informal specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) natural language specifications [3, 23], and is achieved by the use of conventional natural language processing techniques (e.g. see [7]) and a domain (such as information filtering) knowledge base. TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all the aspects of the component, as required by the UMM. The developer provides the necessary implementation for the computational, behavioral, and QoS methods. This process is followed by the QoS validation. If the results are satisfactory (as required by the QoS criteria) then the component is deployed on the network and eventually, it is discovered by one or more headhunters. If the QoS constraints are not met then the developer refines the UMM specification and/or the implementation and the cycle repeats.

4.2 Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, our approach is to translate this natural language specification into a more formal specification using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars, one corresponding to a set of type declarations and the other a set of function definitions operating on those types. These type and function definitions are incorporated into a class which allows for new types to be created.

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. On the other hand, function definitions may be given without precisely defined domains for a more flexible specification approach. This framework consists of a knowledge-base which establishes a context for the natural language text to be used in the specification under a particular domain model, in this case information filtering. This allows the TLG to be translated into internal representations such as predicate logic, the natural representation for TLG, event grammars, or multi-level Java interfaces taking the form of the UMM specification template. For the case study, we may use a TLG class to describe the component structure and functionality as elaborated in the following subsections.

4.2.1 Component Structure Specification

Syntactically, TLG type declarations are similar to those in other languages. Types are capitalized whereas constants begin with lower case letters. The usual primitive types, such as Integer, Float, Boolean, and String are present as are list constructors based upon regular expression notation, e.g. {X}* and {X}+ mean 0 or more and 1 or more occurrences of X, respectively.

The types of the domain component in our information filtering system are defined in the following way in TLG.

```
Component :: DomainComponent; WrapperComponent; RepresentationComponent; ClassificationComponent;
  UserInteractionComponent; HeadhunterComponent; ICB.
DomainComponent :: Name, InformalDescription, Attributes, Service.
Name :: dc.
Attributes :: ComputationalAttributes, CooperationAttributes, AuxiliaryAttributes.
ComputationalAttributes :: InherentAttributes, FunctionalAttributes.
InherentAttributes :: Author, Version, DateDeployed, ExecutionEnvironment,
  ComponentModel, Validity, Structure, Registrations.
FunctionalAttributes :: TaskDescription, AlgorithmAndComplexity,
  Alternatives, Resources, DesignPatterns, Usages, Aliases, FunctionsAndContracts.
AlgorithmAndComplexity :: webcrawling, n^2; ....
Alternatives :: {AlgorithmAndComplexity}*.
Resource :: Architecture, Speed, Load.
Architecture :: uni-processor; multi-processor.
Speed :: Integer.
Load :: Integer.
DesignPatterns :: broker; ....
Aliases :: pro-active agent; ....
FunctionAndContract :: Function, BehavioralContract, ConcurrencyContract.
Function :: ....
BehavioralContract :: Precondition, Invariant, Postcondition.
ConcurrencyContract :: single threaded; maximum number of active threads allowed = Integer; ....
CooperationAttributes :: ExpectedCollaborators, PreprocessingCollaborators, PostprocessingCollaborators.
ExpectedCollaborator :: uic; wc; hc; tc; rc.
PreprocessingCollaborator :: hc; tc.
PostprocessingCollaborator :: rc; uic.
AuxiliaryAttribute :: FaultTolerantAttribute; SecurityAttribute; MobilityAttribute.
FaultTolerantAttribute :: check-pointing versions; ....
SecurityAttribute :: simple encryption; ....
MobilityAttribute :: mobile; not mobile.
Service :: ExecutionRate, ParallelismConstraint, Priority, Latency, Capacity, Availability,
  OrderingConstraints, QualityOfResultsReturned, ResourcesAvailable, ....
ExecutionRate :: Float.
ParallelismConstraint :: synchronous; asynchronous.
Priority :: Integer.
Latency :: AverageRateOfURLCollection.
AverageRateOfURLCollection :: Float.
Capacity :: NumberOfAvailableURLs.
NumberOfAvailableURLs :: Integer.
Availability :: Float.
OrderingConstraint :: Boolean.
QualityOfResultsReturned :: {URL}+.
ResourcesAvailable :: HardwareResources, SoftwareResources.
HardwareResources :: ....
SoftwareResources :: ....
```

The remaining components (e.g., wrapper, representation, etc.) may be described in a similar manner. All domains not specified explicitly in the above example are assumed to be of type String, with the exception of Function which may take the form of an interface definition in a programming language such as Java. Using standard natural language processing techniques [7], the UMM specification may be automatically refined into this TLG specification, with user assistance as needed to clarify ambiguities. The process is facilitated by the presence of a knowledge base which understands the domain of information filtering from the point of view of vocabulary which may be used in making the original UMM specification.

4.2.2 Component Functionality Specification

The second level of the TLG specification is for function declarations. These resemble logical rules in logic programming with variables coming from the domains established in the type declarations. For the Domain Component example, the levels of Quality of Service may be specified as follows.

```
number of urls : size of QualityOfResultsReturned.
average latency : ...
no ranking of urls : ...
simple ranking of urls : ...
advanced ranking of urls : ...
average latency : ...
qos level 1 is novice : number of urls < 50, no ranking of urls,
  AverageRateofURLCollection >= 1 week, average latency >= 2 minutes.
qos level 2 is intermediate : number of urls < 500, simple ranking of urls,
  AverageRateofURLCollection >= 3 days, average latency >= 1 minute.
qos level 3 is expert : number of urls < 1500, advanced ranking of urls,
  AverageRateofURLCollection >= 1 day, average latency >= 5 seconds.
```

Each rule defines how the particular entity is to be computed. As these rules are normally part of a class definition encapsulating a corresponding set of type declarations, each rule has access to the data specified in the type declarations. These natural language like rules may be further refined into a more formal specification, e.g. using event grammars.

4.3 QoS Guarantee of a Domain Component

For the case study, the event grammar to describe the system behavior is given below. The first part is the set of type definitions and the second part is the description of computations over event traces implementing different QoS metrics.

```
exec_syst :: (request_sent | response_received)*
response_received :: (URL_detected | failed)
```

These type definitions describe the types of events which may occur as the system executes. The computations over these events include verification that the number of URL's detected is less than 50 and also the latency (e.g., for all requests for URL's, every response received occurs within 10 units of time). id is an event attribute which associates a unique identifier between query attributes and corresponding responses. Both of these metrics yield Boolean values.

```
CARD [URL_detected from exec_syst] < 50
```

```
Forall x : request_sent from exec_syst
  Exists y : response_received from exec_syst
    id (x) = id (y) & begin_time (y) - end_time (x) < 10
```

4.4 Automated System Generation and Evaluation based on QoS

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available, then the task is to assemble them. Figure 2 shows a process to accomplish this. The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. For example, such a query might be a request to assemble an information filtering system. The natural language processor (NLP) processes the query. It does this aided by the domain knowledge (such as key concepts in the filtering domain) and a knowledge-base containing the UMM description (in the form of a TLG) of the components for that domain. The result is a formal UMM specification that will be used by headhunters for component searches and as an input to the system assembly step. This formal UMM specification will be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS. The framework, with the help of the infrastructure described in Section 2.2.3, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. From these, the developer, or a program acting as a proxy of the developer, selects some components. These components along with the component broker and appropriate adapters (if needed) form a software implementation of the distributed system. Next this implementation is tested using event traces and the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional

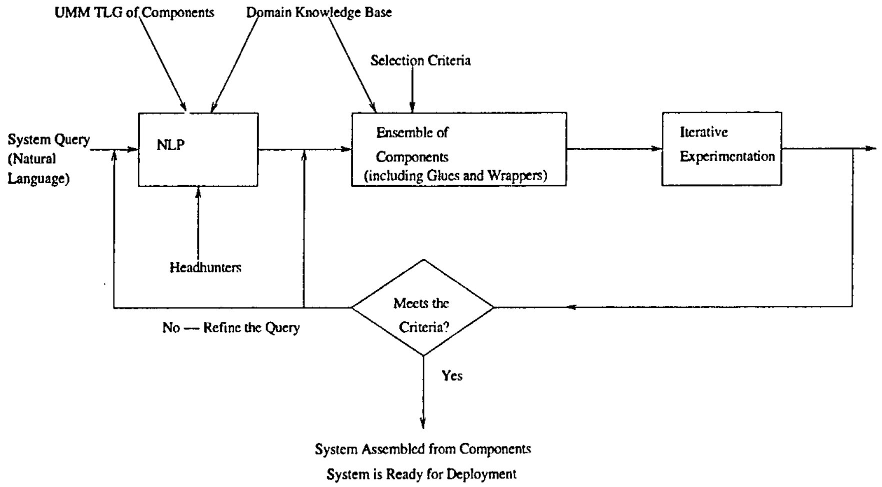

UMM TLG of Components
Domain Knowledge Base
Selection Criteria
System Query (Natural Language)
NLP
Ensemble of Components (including Glues and Wrappers)
Iterative Experimentation
Headhunters
Meets the Criteria?
No — Refine the Query
Yes
System Assembled from Components
System is Ready for Deployment

Figure 2: The Iterative System Integration Process in UMM

components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. Once a satisfactory implementation is found, it is ready for deployment.

5 Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding with the QoS constraints advertised by each component and the collection of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Glue and wrapper technology allows a seamless integration of heterogeneous components and the formal specification of all aspects of each component will eliminate ambiguity while detecting and using these components. The UMM does not consider network failures or other considerations related to the hardware infrastructure, however, these could be integrated into the QoS level of components. The UMM approach to validating QoS is to use event grammar to calculate QoS metrics over run-time behavior. The QoS metrics are then used as a criteria for an iterative process of assembling the resulting system as shown in Section 4.4. UMM also provides an opportunity to bridge gaps that currently exist in the standards arena. Although, the paper has only presented a case study from the domain of distributed information filtering, the principles of UMM may be applied to other distributed application domains. Future work includes refinement of the UMM feature thesaurus and methods for translating UMM specifications into Two-Level Grammar, refining the head-hunter mechanism, developing Quality of Service metrics for components and systems, and development of generation mechanisms for domain-specific component reuse.

References

[1] Auguston, M. A Language for Debugging Automation. In *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering*, pages 108–115, 1994.

[2] Beugnard, A., Jezequel, J., Plouzeau, N. and Watkins, D. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[3] Barrett R. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia*, pages 24–30, January 2000.

[4] California Institute of Technology. *Caltech Infospheres On-line Documentation, URL:- http://www.infospheres.caltech.edu/*, 1998.

[5] Fox, G. The Document Object Model Universal Access Other Objects CORBA XML Jini JavaScript etc. *http://www.npac.syr.edu/users/gcf/msrcobjectsapril99*, 1999.

[6] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83–86, 2(2), 1998.

[7] Jurafsky, D. and Martin, J. H. *Speech and Language Processing.* Prentice Hall, 2000.

[8] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S. and Cybenko, G. Agent TCL: Targetting the Needs of Mobile Computers. *IEEE Internet Computing*, pages 58–67, 1(4), 1997.

[9] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R. and Kin, B. K. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of RSP 2001, the 12th International Workshop on Rapid System Prototyping*, 2001.

[10] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.

[11] Microsoft Corporation. *DCOM Specifications, URL:- http://www.microsoft.com/oledev/olecom*, 1998.

[12] Object Management Group. XML Metadata Interchange. Technical report, Object Management Group Document No. ad/98-10-05, October 1998.

[13] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.

[14] Object Management Group. Meta Object Facility (MOF) Specification, Version 1.3. Technical report, Object Management Group, March 2000.

[15] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01, February 2001.

[16] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.

[17] Raje, R. UMM: Unified Meta-object Model for Open Distributed Systems. In *Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000)*, 2000.

[18] Rogerson, D. *Inside COM.* Microsoft Press, 1996.

[19] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[20] Siegel, J. *CORBA Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[21] The Globus Project. *Globus Website, URL:- http://www.globus.org/*, 2000.

[22] University of Virginia. *Legion Project, URL:- http://www.cs.virginia.edu/ legion*, 1999.

[23] van Wijngaarden, A. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.

Enhancements & Extensions of
Formal Models for Risk Assessment in Software Projects

Michael R. Murrah
mrmurrah@nps.navy.mil

Luqi
Luqi@cs.nps.navy.mil

Craig S. Johnson
csjohnson@dcmdw.dcma.mil

Naval Postgraduate School
2, University Circle
Monterey, CA 93943 USA

Abstract

Over the past 40 years limited progress has been made to help practitioners estimate the risk and the required effort necessary to deliver software solutions. Recent developments improve this outlook, one in particular, the research conducted by Juan Carlos Nogueira [1]. Dr. Nogueira developed a formal model for risk assessment that can be used to estimate a software project's risk when examined against a desired development time-line. This model is based on easily obtainable software metrics. These metrics are quantifiable early in the software development process.

Dr. Nogueira developed his model based on data collected from a series of experiments conducted on the Vite'Project simulation [2]. This unique approach provides a starting point towards a proven formal model for risk assessment, one that can be applied early in the software development lifecycle. Approaching software risk estimation has never previously been successfully accomplished in this manner.

The proposed research will provide definitive evidence that software risk assessment can be conducted early in software development using quantifiable metrics and simple techniques. Enhancements will be made to Dr. Nogueira's model, based on calibrations against post-mortem projects. These enhancements will result from many threads of research; extension of input metrics, increased number of simulation runs, simulation scenarios based on actual projects, and the introduction of a "gearing factor". Ultimately, the research will yield an improved risk assessment model, one that has been validated against thousands of post-mortem projects, having applicability to any software development activity.

1. Introduction

The current state of the art techniques of risk assessment rely on checklists and human expertise. This constitutes a weak approach because different people could arrive at different conclusions from the same scenario. The difficulty of estimating the duration of projects applying evolutionary software processes adds intricacy to the risk assessment problem.

2. Dr. Nogueira's Risk Assessment Model

Dr. Nogueira's research introduces a formal method to assess the risk and the duration of software projects automatically, based on measurements that can be obtained early in the development process. The method has been designed according to the characteristics of evolutionary software processes, and utilizes quantifiable indicators such as efficiency, requirement volatility and complexity. The formal model, based on these three indicators estimates the duration and risk of evolutionary software processes. The approach introduces benefits in two fields:

a) Automation of risk assessment.
b) Early estimation methods for evolutionary software processes.

Dr. Nogueira developed four software risk estimation models that show great promise in determining a software projects' associated risk early in the software development life cycle. The models accomplish early estimation by utilizing a set of quantifiable metrics that can be collected from the beginning of project development. In actuality, the requirements volatility metric is an estimation during the first development cycle and during subsequent development cycles is quantifiable. After each iteration of software development, the required input metrics can be applied to the model in order to reduce the error in the model's results.

The minimum required input metrics, to support risk assessment, required for Dr. Nogueira's estimation model are the following:

a. Efficiency (EF) – The efficiency of the organization can be measured observing the fit between people and their roles [1]. Dr. Nogueira's research indicates that the efficiency of an organization can be directly calculated by computing the ratio of direct time (working and correcting errors) divided by the idle time (time spent without work to do).

b. Requirements Volatility (RV) – Requirements volatility expresses how difficult the requirement elicitation process is. The requirements volatility is obtained by the following formula [1].

Requirements Volatility = Birth Rate Percentage + Death Rate Percentage

Birth Rate Percentage (BR%) = the percentage of new requirements incorporated in each cycle of the software evolution process as calculated by:

BR% = (New Requirements / Total Requirements) * 100 percent

Death Rate Percentage (DR%) = the percentage of requirements that are dropped by the customer in each cycle of the evolution process as calculated by:

DR% = (Deleted Requirements / Total Requirements) * 100 percent

c. Complexity (CX) – Complexity has a direct impact on quality because the likelihood that a component fails is directly related to its complexity [1]. The complexity metrics can be determined in two forms: large granular complexity and fine granular complexity. These two forms of complexity can be directly determined from software specifications written in the Prototype System Description Language (PSDL) [3].

Large Granular Complexity (LGC) expresses the relational complexity of the system as a function of the number of operators (O), data streams (D), and types (T)

LGC = O + D + T

Fine Granular Complexity (FGC) expresses the relational complexity of each operator in the system and is a function of the fan-in and fan-out data streams related to the operator [1]. For the purposes of the completed research and our notion of future research, the FGC metric is too specialized; our efforts concentrate on just the representation of the LGC.

FGC = fan-in + fan-out

Software developers can utilize Dr. Nogueira's four models to assess either the development time required to develop a project or determine the associated probability of completing a software project given the project's duration.

3. Previous Validation Research

In this section of the paper we present the results of validation attempts when using Dr. Nogueira's estimation models. The first is a result of the research conducted by Dr. Nogueira in his initial research and supplies data from simulations and comparisons to one project. The second validation endeavor is the results of research conducted on two additional projects [5].

3.1 Dr. Nogueira's Validation

In conducting his research, Dr. Nogueira derived some initial conclusions with the models. The simulations showed that the three risk factors observed during the causal analysis (efficiency, requirements volatility, and complexity) have compound effects over the three parameters of the Weibull distribution [1].

Dr. Nogueira illustrates the results of the models against 16 simulated projects. Each model derives an increasing degree of accuracy based on: metrics from the three risk factors, Weibull cumulative density function, and the derivation of the time.

Models 1-2. Model 1 can be used when the requirements volatility is small. Model 2 considers the three factors (EF, RV, and CX), but neglects the combined effect of EF and RV.
Figure 1 illustrates the results of the models which were calculated using 95% of confidence (p=0.95). Note the errors as vertical segments between the estimated and real values.

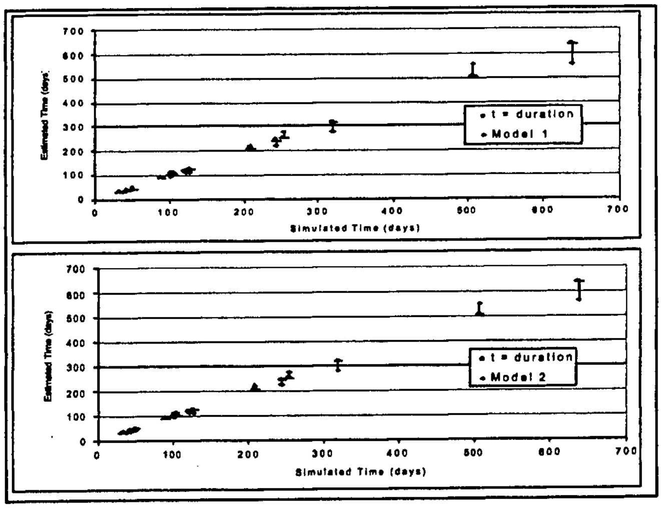


Figure 1. Scatter Plot of Models 1-2

Model 3. Model 3, illustrated in Figure 2, considers the three factors as well as the combined effects of EF and RV. The analysis of variance shows that the samples obtained from the simulations and the samples obtained from the estimates using Model 1, 2 or 3 cannot be statistically differentiated.

Another interesting result is that the errors remain in the range of±15% for all of the scenarios. This result is interesting if we compare it with the results of COCOMO (±20% in the best cases). Barry Boehm in reference to the validation of COCOMO said, "In terms of our criterion of being able to estimate within 20% of projects actuals, Basic COCOMO accomplishes this with only 25% of the time, Intermediate COCOMO 68% of the time, and Detailed COCOMO 70% of the time." [4].

Model 4. Model 4, Figure 2, can be used for any range of complexity and requirements volatility, and considers the three factors, their combined effects, and the following a priori assumptions:

- A project with 0 LGC will take 0 days
- α, β, and $\gamma > 0$

- If RV increases the p(x<=t) decreases
- If CX increases then p(x<=t) decreases
- If EF increases then p(x<=t) increases

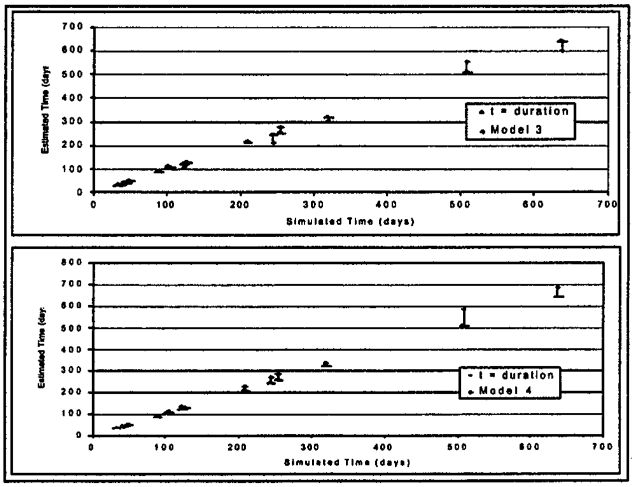


Figure 2. Scatter Plot of Models 3-4

The scatter plot in Figure 2 compares the simulated times versus the estimated times. Most of the errors are overestimations and the duration of the project has no effect over the percentage of error. Model 4 is conservative. The maximum overestimation error was less than 16% and the maximum underestimation was less than 4%.

Model 4 gives a good estimation for projects between 4,000 and 20,000 LGC (128 and 640 KLOC of Ada). The estimation seems to be too optimistic for projects smaller than 1000 LGC but it is quite good for larger projects. To verify the model Dr. Nogueira used a real project consisting of 1836 LGC developed in 1.5 years by the Uruguayan Navy[1]. Model 4 predicts 17 months instead of 18 months, the actual development time.

3.2 Additional Project Validation

Project A [5]. We used Nogueira's Model 4 to calculate the probability of completion curve for the projects. For consistency, we used working days, defined as 22 days per month, the same as used in the original Nogueira model.

The model predicted that the minimum time, in days, necessary to have a probability of completion of 100% is approximately 260 working days. When compared to the actual time it took, which was 336 working days, the model predicted completion sooner. The model predicted 76 working days less, or a 22.6% delta.

$$\mathbf{(1 - (260 / 336))\,(100) = 22.6.}$$

At this point, with 22.6% variability, we decided to investigate and see what the original estimated completion date was from project records. The original estimation was 200 working days, with the project schedule slipping 136 working days for build 3. The developer missed the original completion estimation by 40.5%.

[1] SIMTAS a simulator for war gaming with 75,240 lines of code

(1- (200 / 336)) (100) = 40.5.

The Nogueira model missed the developer's original estimate by 23.1%

(1-(200 / 260)) (100) = 23.1

Does this mean that the Nogueira model is too optimistic as are most developers' estimates, or is it a better fit? This data point leaves us with an inconclusive position as to the validation of the model against the first project. It appears that there is a difference when using real projects with real data versus simulated project data, and this reflects what the real world is – unpredictable.

Project B [5]. We used Dr. Nogueira's Model 4 to calculate the probability of completion curve for Build 2 using; BR=2.59, DR=3.04, RV=5.63, O=2544, D=4010, T=1003. The model predicted Impossible.

Actual time for build 2 took from 4/24/00 until 7/10/00 or 68 working days at 22 working days a month. We believe this inconsistency is due primarily because the calculation for the LGC count is based on all six Computer Software Configuration Items (CSCI). Core functionality on three CSCIs; CSCI-A, CSCI-B, and CSCI-C had been previously developed and validated. However, the builds during this period, involved addition of functionality to the following CSCIs: CSCI-D, CSCI-E, and CSCI-F. That is, build 2 was modifying only a portion of the total software system code, but the LGC data gives a view of all six CSCIs combined.

The available data was not broken down into separate CSCIs, nor does it, post-mortem, identify the code that was being worked in a previous software release. We cannot fault the developer for not collecting metrics for research concepts that they are not aware of, nor do we believe that this type of data collection is a requirement of CMM level 3.

A finding of this research is the need to adjust the CX when applying the Nogueira model to evolved projects that are developing or enhancing only a portion of their CSCIs.

Additionally, this project did not utilize a lower case tool such as Rational Rose. We believe use of such a tool is essential when attempting to apply the Nogueira formal model, as it provides the capability to collect detailed information, over the software development lifecycle, that can later be extracted and used for input to the Nogueira model metrics.

4. Issues with Dr. Nogueira's Risk Assessment Model

Applying Dr. Nogueira's risk assessment model, in its current form, presents a number of issues that must be resolved before substantial progress can be achieved validating the model's results. The first issue and most notable draw back when using Dr. Nogueira's risk assessment model is limited confidence that the model provides valid results. This is due to three factors: the limited amount of time that the model has been in existence, the model has not been exercised on a wide base ofreal world projects (completed or on-going), and the fact that the model was developed using simulation techniques. The first factor noted can only be dealt with in the passage of time. However, this research will exploit a unique opportunity to impact the latter two issues.

Although Dr. Nogueira's research shows promise in estimating the associated risk when developing software systems, the model has not been significantly exercised beyond theoretical simulation. Three "real world" projects to date have been applied against the estimation model [1], [5]. It should be noted that all three of these projects were exercised post-mortem. Model validity has not been demonstrated in the context targeted by the model's original design, estimating risk early in a software project's life cycle.

A second issue that exist when using Dr. Nogueira's risk assessment model is the required input metrics. This issue is a double-edged sword. A major attraction to using Dr. Nogueira's model are these metrics.

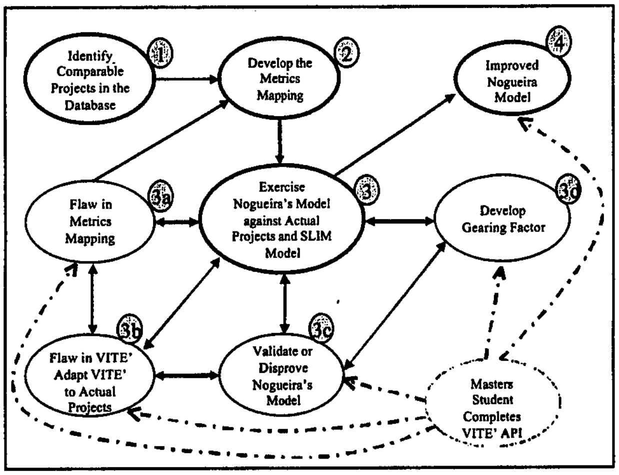
They are determined in a definitive, quantifiable manner and can be derived extremely early in the software development process [1], [6]. However, these metrics are quite unique. Currently, outside of the academic environment, it is not common practice to collect these unique metrics in the required form to utilize Dr. Nogueira's risk assessment model.

In order to establish confidence in the usefulness and accuracy of Dr. Nogueira's risk estimation model, the model must be exercised against numerous projects. It would be ideal, and perhaps over time, to exercise the model according to it original design; early in the software development cycle. However, the next logical step is to continue to exercise the model in a post-mortem basis. Before this can be accomplished, two things need to happen: First correlations must be determined between Dr. Nogueira's required metrics and metrics that are frequently collected in historical project databases. By establishing metrics correlations, the model can be exercised against an additional project base helping address the second factor of problem one. And second, a method other than the use of PSDL to generate O, D and T metrics counts must be developed. Dr. Nogueira's model was based on using PSDL to automatically scan and generate counts for O, D, and T input to his model. It is unlikely that PSDL was used on any programs that we have post-mortem data on.

The final problem associated with Dr. Nogueira's risk assessment model is the configuration of the Vite'Project simulation. Dr. Nogueira developed the configuration of Vite'Project using Organizational Consultant expert system. Fictitious software engineering organizations were developed to represent the typical software development department. Based on the results of establishing fictitious CMM level 2 and level 3 organizations, the Vite'Project was calibrated. Calibrating the simulation in this manner, could yield different results than calibrating the simulation with actual information derived from real projects. If Dr. Nogueira's model can be verified by reprogramming the Vite'Project configuration this would provide additional assessment to the third factor of problem one.

5. Proposed Research

The proposed research will expand the efforts of the previous validation effort. Figure 3 outlines the research approach.



Identify Comparable Projects in the Database
1
Develop the Metrics Mapping
2
Improved Nogueira Model
4
Flaw in Metrics Mapping
3a
Exercise Nogueira's Model against Actual Projects and SLIM Model
3
Develop Gearing Factor
3d
Flaw in VITE' Adapt VITE' to Actual Projects
3b
Validate or Disprove Nogueira's Model
3c
Masters Student Completes VITE' API

Figure 3. Phases of Research

Phase one: During phase one of the research, post-mortem projects will be identified whose characteristics are similar to the characteristics of the three projects previously exercised against Dr. Nogueira's risk assessment model. This affords the opportunity to begin with a baseline before proceeding to future phases.

Phase two: This is the most challenging phase of the research and we hypothesize that this phase will consume the majority of the available resources. In this phase, detailed analysis is conducted against the available metrics that have been collected on the projects established during phase one. Correlations are determined in the available data against the three metrics that are necessary when utilizing Dr. Nogueira's model. Upon completion of this phase, when a suitable "metric map" has been developed, research can continue to phase three. The intent of the metric map is to provide a common platform to exercise Dr. Nogueira's model using metrics that were not originally collected for this purpose.

Phase three: Once a suitable metric map has been established, research continues by exercising Dr. Nogueira's model against the set of post-mortem projects determined in phase one. This phase is essential to establish confidence in the results produced when using Dr. Nogueira's model. Additionally during this phase, another risk assessment method is introduced, Quantitative Software Management's® (QSM) SLIM, to help in the validation process. Essentially, there will be a comparison of three artifacts: the recorded project performance, the estimated project performance using Dr. Nogueira's model, and the estimated project performance as determined by QSM's SLIM. An assumption during this phase will be the accuracy of QSM's SLIM. Of course, if the expected results are not achieved during this phase, additional research must be performed to determine the cause of the variance.

Phase three (a): One potential cause of the variance observed during phase three could be a flaw in the metric map determined during phase two. Continued research will be conducted to modify the mapping and eventually minimize the chance that the metric map is the source of the deviation.

Phase three (b): Another factor that can influence deviation between the actual project data, Dr. Nogueira's estimation model, and QSM's SLIM estimation model is the original configuration used to establish project scenarios in the Vite'Project. Organizational Consultant expert system was used to establish fictitious software engineering organizations. Research may indicate that reprogramming the Vite'Project with actual information from software development organizations could yield different results in the Vite'Project simulation. This was a fundamental factor in the development of Dr. Nogueira's research. A substantial change in the simulated results could require extensive rework of Dr. Nogueira's model.

Phase three (c): Finally, after exhausting Phases three (a & b), research may lead to examination of Dr. Nogueira's model with closer scrutiny. If deviation continues to present itself when conducting phase three, we may have essentially resort to "ground zero" to establish potential conflicts. *It should be noted that phases three (a, b, & c) should not be considered mutually exclusive. Research could indicate that partial modifications are required in all three sub-phases.*

Phase three (d): Dr. Nogueira's risk assessment model is perfectly suited for any evolutionary software process because it follows the same philosophy [1]. Dr. Nogueira presents no hypothesis of the model's validity when the model is exercised outside of this domain. Once phase three is accomplished and confidence has been established against the set of projects determined during phase one, the model can be exercised against additional projects, from different industry sectors and different software development methodologies. This may require the development of what we are calling a "gearing factor". In this research, the use of this term is intended to represent a value that is multiplied by the results determined in Dr. Nogueira's model, adjusting the results for the new domain. In some cases the model may provide suitable results without the use of a gearing factor, other domains and development methodologies may require this adjustment due to the unique nature of the software's development.

Phase four: Phase four of the proposed research is the culmination of all of the proposed research. This phase delivers the improved Nogueira model. A caveat to this phase and all of the sub-phases conducted during phase three is the introduction of the Vite'Project API. This automated tool will improve the statistical significance obtained when utilizing the Vite'Project simulation, greatly increasing the number of simulation runs provided by the simulation.

6. Validation

We propose to validate our research by conducting controlled experiments against post-mortem projects. QSM, founded in 1978 by Larry Putnam, has collected and maintained an extensive database of over 5,000 software projects [7]. Experiments can be conducted, utilizing the available software metrics from QSM's database, that correlate the required metrics in Dr. Nogueira's model. This will afford our research the means to evaluate actual projects against Dr. Nogueira's model.

Another source of validation is obtained by configuring Vite'Project with actual software project development information. As previously mentioned, Vite'Project scenario's were originally established by the creation of fictitious software development organizations. Different results could be derived from simulations configured according to actual projects.

Finally, we propose to increase the statistical significance of Dr. Nogueira's software risk assessment model. We can accomplish this by increasing the simulation runs of each scenario through automation via the Vite' API when available.

7. Conclusion

This research introduces a research plan to validate a formal risk assessment model for software projects based on probabilities and metrics automatically collectable early in the project. The approach enables a project manager to evaluate the probability of success of the project very early in the life cycle. For more than twenty years the estimation standards (COCOMO 81, COCOMO II, Putnam) have been characterized by a common limitation: the requirements should be frozen in order to make estimations. This promising model removes this important limitation, facing the reality that requirements are inherently variable.

The problem of risk assessment for projects has been treated as unstructured. Research shows, and experiments will prove, a structured method to solve the problem based on metrics automatically collected from the project baselines. This contribution impacts the software engineering state of the art, as well as risk management in general. These metrics measure three risk factors identified in the research: complexity, requirements volatility, and efficiency. The subjectivity issue characteristic of previous research has been eliminated. Any decision-maker will arrive at the same estimates, independently of his or her expertise.

Finally, current research is based on simulations and a small set of real projects. It is desirable to collect and analyze metrics and completion times of a larger set of real software projects to confirm and refine the models. Our research will provide the missing elements from the models, validation, enhancements, and extensions.

References

[1] Nogueira J.C., *A Formal Model for Risk Assessment in Software Projects*. PhD Dissertation. Naval Postgraduate School. Monterey, California. 2000.

[2] The Vite'Project Handbook. Vite©. 1999.

[3] Berzins, V. and Luqi. *Software Engineering with Abstractions.* Addison-Wesley, 1990.

[4] Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

[5] Johnson, C. S., Piirainen R. A. *Application of the Nogueira Risk Assessment Model to Real-Time Embedded Software Projects.* Masters Thesis. Naval Postgraduate School. Monterey, California. March 2001.

[6] Nogueira, J.C., Luqi, Bhattacharya, S. A Risk *Assessment Model for Software Prototyping Projects.* Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on , 2000 Page(s): 28 -33

[7] SLIM MasterPlan User's Guide. QSM® March 2001.

Visual Meta-Programming Notation[1]

Mikhail Auguston[2]
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943 USA
auguston@cs.nps.navy.mil

Abstract

This paper describes a draft of visual notation for meta-programming. The main suggestions of this work include specialized data structures (lists, tuples, trees), data item associations that provide for creation of arbitrary graphs, visualization of data structures and data flows, graphical notation for pattern matching (list, tuple, and tree patterns, graphical notation for context free grammars, streams), encapsulation means for hierarchical rules design, two-dimensional data-flow diagrams for rules , visual control constructs for conditionals and iteration, default mapping rules to reduce real-estate requirements for diagrams, and dynamic data attributes.

Two-dimensional data flow diagrams improve readability of a meta-program. The abstract syntax type definitions for common programming languages and related default mappings (parsing and de-parsing) provide for a practically feasible reuse of those components.

1 Introduction and objectives

Meta-programs are programs manipulating other programs. Typical applications include compilers, interpreters, source code static analyzers and checkers, program generators, and pretty-printers. Domain-specific language implementation and rapidly evolving generative programming [9] are the latest examples of developments in this domain. The complexity and sophistication of meta-programs may be quite significant, so the readability and maintainability become an issue.

Compiler and generator design is a domain that has been studied extensively. There is a pretty good understanding of what to do and how to do it, especially for front-end design, and a lot of domain-specific software design templates are accumulated in literature. The following domain features are among the most common for language processor design.

- Use of context-free grammars to specify syntax and serve as a basis for parser design.
- Intermediate representation of the input in the form of an abstract syntax tree. The importance of different tree data structures is recognized in general for this problem domain.
- Typically, the main components of a language processor are very hierarchical and structured along the structure of data (recursive descent parser is an excellent example of this feature). In other words, language processors are heavily data-based applications.
- It appears that the most commonly used data structures include trees, lists, stacks, tables, and strings.
- The architecture of a language processor in most cases can be represented as a data flow between components (e.g., the famous compiler data flow diagram on the page 13 of the "Dragon Book"[1]).
- The notion of an attribute associated with the data item, and attribute dependency and propagation schemes are of a great relevance (the attribute grammar framework captures some of the essential static checking needs; the data flow analysis performed for the optimization stage in a compiler may be considered as an attribute propagation over the program graph).

[1] This research was supported in part by the U. S. Army Research Office under grant number 40473-MA-SP.

[2] On leave from New Mexico State University, USA

- Tree (and graph) traversal and transformation is a common template for optimization and code generation tasks.
- Pattern matching (e.g., with respect to regular expressions or context-free grammars) may be a useful control structure for this problem domain.

These considerations and experience with the compiler writing tools RIGAL[2][3], lex and yacc[11], and ELI[10] contributed to this work. Data-flow paradigm is quite natural for meta-programming domain since it is heavily data dependent, and consequently, the graphical notation for data-flow diagrams could be appropriate. This should be integrated with visualization of typical data structures, pattern matching, and encapsulation to provide for well-structured, hierarchical programs. Data-flow diagrams are most commonly used to represent dependencies between data and processes in visual programming languages, for instance, in LabVIEW[5] and Prograph[8].

Two-dimensional diagram notation could significantly improve readability of meta-programs. Some of these ideas have been explored in our previous work[4].

The main suggestions of this work are as follows:

- specialized data structures (lists, tuples, trees),
- data items associations that provide for creation of arbitrary graphs,
- visualization of data structures and data flows,
- graphical notation for pattern matching (list, tuple, and tree patterns; graphical notation for context free grammars and streams),
- encapsulation means for hierarchical rules design,
- two-dimensional data-flow diagrams for rules,
- visual control constructs for conditionals and iteration,
- default mapping rules to reduce screen real-estate requirements for diagrams,
- dynamic (Last #rule $attribute) and static (via associations) data attributes,
- data-flow notation that assumes potential parallelism in the data processing,
- abstract syntax type definitions for common programming languages and related default mappings (parsing and deparsing) that provide for a practically feasible reuse of those components.

2 Constructs

This paper was not intended to give a complete and precise syntax and semantics of the visual language. At this point it is rather a notation that will be upgraded to programming language status after the implementation effort is completed. A (simplified) example of a compiler from a small subset of Lisp (called MicroLisp) to the C language will be used to present the main ideas. Figures 3– 7 present several annotated parsing and code generation rules of the MicroLisp to C compiler. Appendix A contains the MicroLisp context-free grammar and an example of a program.

2.1 Data flow diagrams

Detailed rationale for data-flow diagram notation and a survey of related work can be found in a previous paper[4]. Briefly, a meta-program is rendered as a two-dimensional data flow diagram that visualizes the dependencies between data and processes. Diagrams actually are similar to the notion of procedure in common programming languages. A diagram represents a single function called a rule, and rule calls may be recursive. The data-flow diagram supports the possibility of parallel execution of threads within the rule.

The data-flow paradigm is closely related to the functional programming paradigm [7] and shares with that paradigm referential transparency and good correspondence between the source code (the diagram) and the order of program execution.

Each diagram represents a single function with several inputs and outputs. At the top of a diagram a signature of a rule provides the rule name and types of its inputs and outputs. Besides data items, the diagram may also contain control structures, such as other rule calls, conditional data flow switches, and iterative constructs [4]. All of those constructs are illustrated in the MicroLisp examples.

The rectangular boxes in our notation denote values, and circles and ovals denote patterns, that could be matched with data objects.

2.2 Types

Type represents a set of values (or objects). Basic predefined types include `char` (characters) and `int` (integers). There is also a universal type `ANY` (which is a super type for any type) and the minimal type `NULL` (which is a subtype of any other type and contains a single value `Null` representing also an empty list or tuple).

Aggregate types are ordered tuples of heterogeneous objects, which are useful for abstract syntax representation, and lists (sequences of homogeneous objects that could be dynamically augmented). Extended BNF notation may be used to define tuple types. To a large degree the type system is similar to the type mechanisms in VDM [13] and Refine[12].

Example of a tuple type definition.

```
prog::= function-def* expression
```

This establishes that an object of the type prog is a sequence of zero or more objects of the type function-def followed by an object of the type expression. This could be considered as an abstract syntax representation for the MicroLisp program level. Notice that ordered sequence of objects of the type function-def is nested within an object of the type prog.

Example of a list type definition.

```
text :: [char]
```

There is a predefined list type `id:: [char]`, which stands for a set of character strings that are valid identifiers.

Example of a type definition with several alternatives (union type).

```
expr :: int | id | simple-expression
```

This effectively declar es that types int and id are subtypes of expr in the scope of this definition.

Appendix B presents some of the type definitions for the MicroLisp example.

2.3 Default mappings

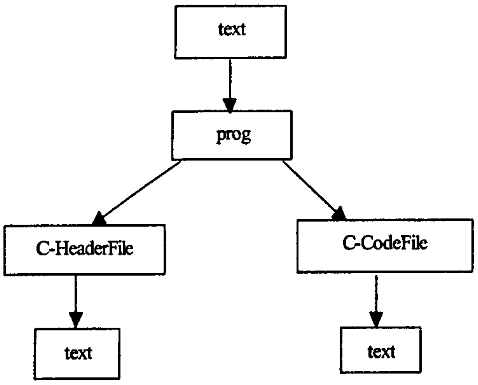


Figure 1. The top level data flow diagram for MicroLisp to C compiler

Certain rules may be declared as default mappings. It means that corresponding rul e calls are optional in the diagrams, and input and output data boxes may be connected directly. This helps to save some screen real estate and to make diagrams less crowded and more readable. Typically default mappings may be introduced for text -to-abstract syntax (parsing) and for abstract syntax-to-text mappings (de-parsing, or abstract syntax -to-concrete syntax mappings).

Yet another kind of default mappings is associated with concatenation operations for tuples and sequences. In fact this is a composition of parsing and de -parsing default mappings applied in the context of (visualized) concatenation. See MicroLisp generation rules for examples (Figures 6 -7).

Definitions of abstract syntax types for common programming languages and related parsing and de -parsing default mappings may be valuable assets for reuse.

Default mappings also open the road for "lightweight" inference. For example, suppose that type A is defined as follows:

```
A :: B | C
```

and there are default mappings B -> D and C -> D, then it is possible to derive a default mapping for A -> D. This example actually addresses the polymorphism issue in our lightweight type system. Similar inference rules could be developed for other aspects of type system based on transitivity of subtype relation.

2.4 Associations

Data objects may be associated with other data objects. Each of those objects may have other associations as well. Associations are not a necessary part of the type definition (although they could be included in the type definition as well) and are rather optional named attributes of particular objects. Associations may be used to create arbitrary graphs from objects. The following picture on Figure 2 illustrates the creation of a graph structure via associations from three data objects. Association is not symmetric. According to the following diagram object A has been associated with an attribute B via an association named ab, object B with C via bc, and C with A via ca.

Associated objects are retained when the host objects are the source and target in an identical transformation (plain arrow connecting data boxes of the same type) or are passed as inputs and outputs of rule calls. A special built -in rule #COPY creates a copy of an object but retains only those components declared in the type definition. Associated objects could be retrieved by pattern matching. For instance, on the right -hand diagram on Figure 2, object C (belonging to the associations established in the previous example) may be passed as input, and an access to objects B and A can be obtained via pattern matching (circles denote object patterns here). Notice that the direction of association arrow indicates the access path from the host object to the attribute object. The association mechanism may be useful to simulate attribute -grammar–like attribute propagation in ensembles of objects, to represent collections of objects as graphs, to implement symbol tables (where identifiers may be represented as associations names), and so on.

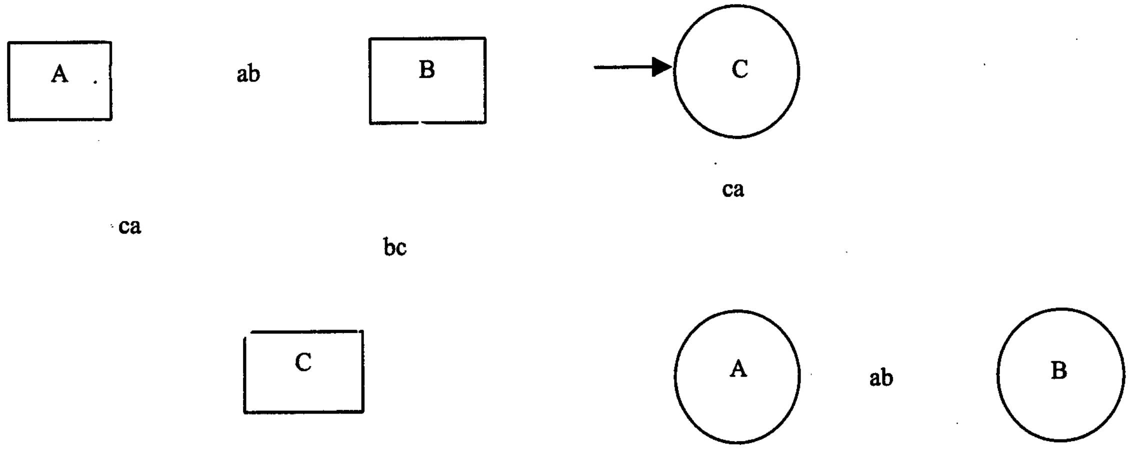


Figure 2. Construction of associations between objects and retrieval of them using pattern matching

2.5 Patterns and streams

Data object patterns are used to visualize structure of objects in order to provide access to object components and associated objects. An object pattern may be placed in any part of the data flow and is matched with the object connected to the pattern input.

If pattern matching is successful the input object is passed downstream. If pattern matching fails, the entire diagram execution fails, and the diagram sends to its outputs a default value Null, unless the pattern has been provided with the 'Failed' output route. See MicroLisp rules in Figures 3-4 for examples.

If a rule's input is a list, patterns applied to this input may be chained in a sequence (using thick gray arrows) to be applied consecutively. This pattern sequence consumes as many objects from the stream as it can successfully match. The notion of stream corresponds to the sequence in RIGAL language [2][3], and semantics of pattern matching is derived from RIGAL's pattern matching semantics. See MicroLisp parsing rules for example (Figures 3-5).

Rules can create output streams of objects as well.

2.6 States and dynamic attributes

Rule may have states – objects that persist while rule instance is active and can be updated by assignment operators within the rule or from other rules called from this rule. This mechanism could be actually considered a macro extension for diagram notation when a corresponding state object is passed to the called rules as an additional parameter and returned back to the callee as an additional output. States have names starting with the $ symbol, e.g. $X. The reference to the rule's #A state $X has a form Last #A $X. When referred within the rule #A, the prefix Last #A can be dropped. See Figures 4-5 for examples.

3 Examples of MicroLisp to C compiler rules

The following diagrams present three top level parsing rules and two top level generation rules for MicroLisp -> C compiler. They illustrate most of the notations discussed above. Additional annotations provide more specific details and discussion. Those rules are deployed according to the data flow diagram on Figure 1 and default mappings in Appendix B.

3.1 Parsing

The source code of MicroLisp program is represented as a stream of characters. It is assumed that there is a lexical component that filters out comments, spaces, tabs, end-of-line characters from the stream before it is fed to the parsing rules.

```
#program: Stream [char]-> prog, Stream [message]
   state $func-list: [id]   -- updated by #func-def
```

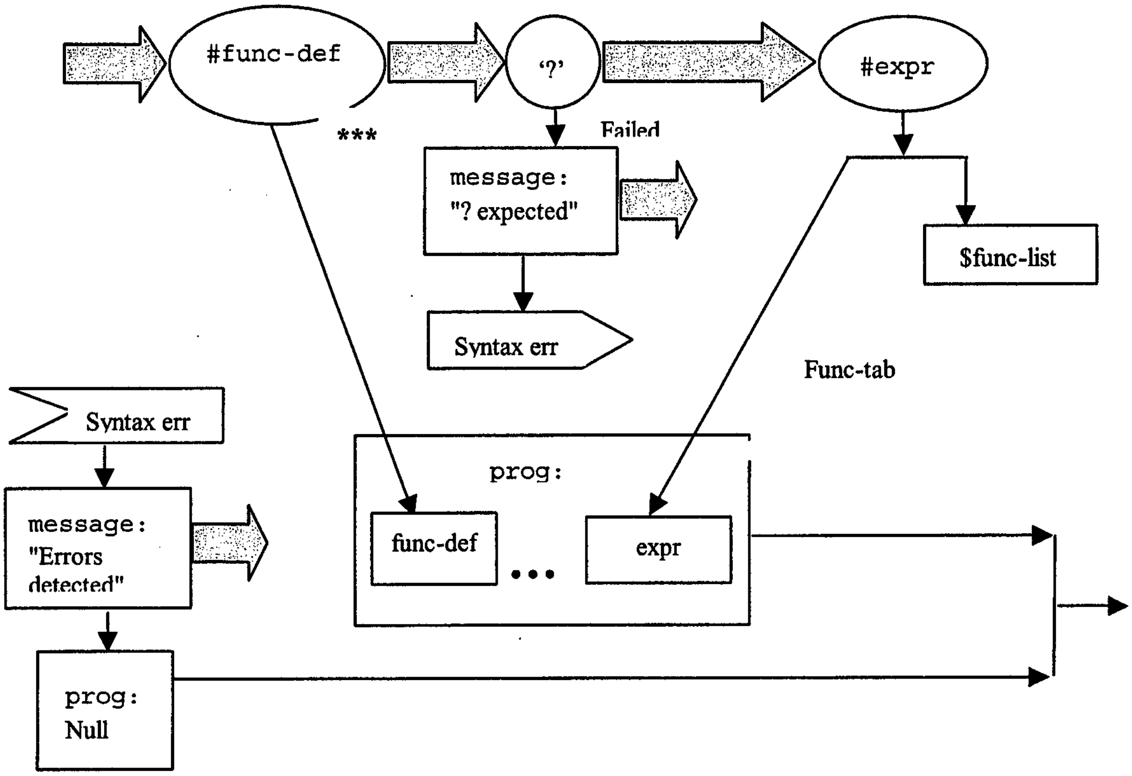


Figure 3. Parsing rule for the grammar rule
`program ::= func-def * '?' expression`

Annotations for the rule #program

- This rule has a state $func -list which will be gradually updated by the rule #func -def calls (see Figure 4). At the end of parsing, object $func-list will be added as an attribute (via association with the name Func -tab) to the resulting object of the type prog. The box containing $func -list has a dummy input of the type ANY, which is activated when the last pa t-tern #expr terminates with success. This ensures the timing when the state value is picked up for the association o pera-tion.
- The rules #func-def and #expr are used as patterns. If pattern matching encapsulated in these rules is successful, the rules also are successful and return values, which are used to assemble the return value of the rule #program.
- If pattern matching for the pattern '?' fails, the entire rule #program also fails and returns object Null, but before it happens two messages will be sent to the output stream. Markers labeled 'Syntax err' are used to prevent a mess with arrow intersections.
- A data flow fork denotes duplication of the data item sent to two or more threads.
- Nesting boxes and forwarding output of pattern rules of the types func-def and expr inside the resulting box of the type prog provide an intuitive visualization for the tuple constructor.
- The application of pattern #func -def may be repeated zero or more times (indicated by the ellipsis '***'), and it is sy n-chronized with the tuple constructor (as the box of the type func-def in the resulting prog box is also accompanied by an ellipsis).

```
#func-def: Stream [char]-> Func-def, Stream [message]
           state $param-list: [id]     -- used in #expr
```

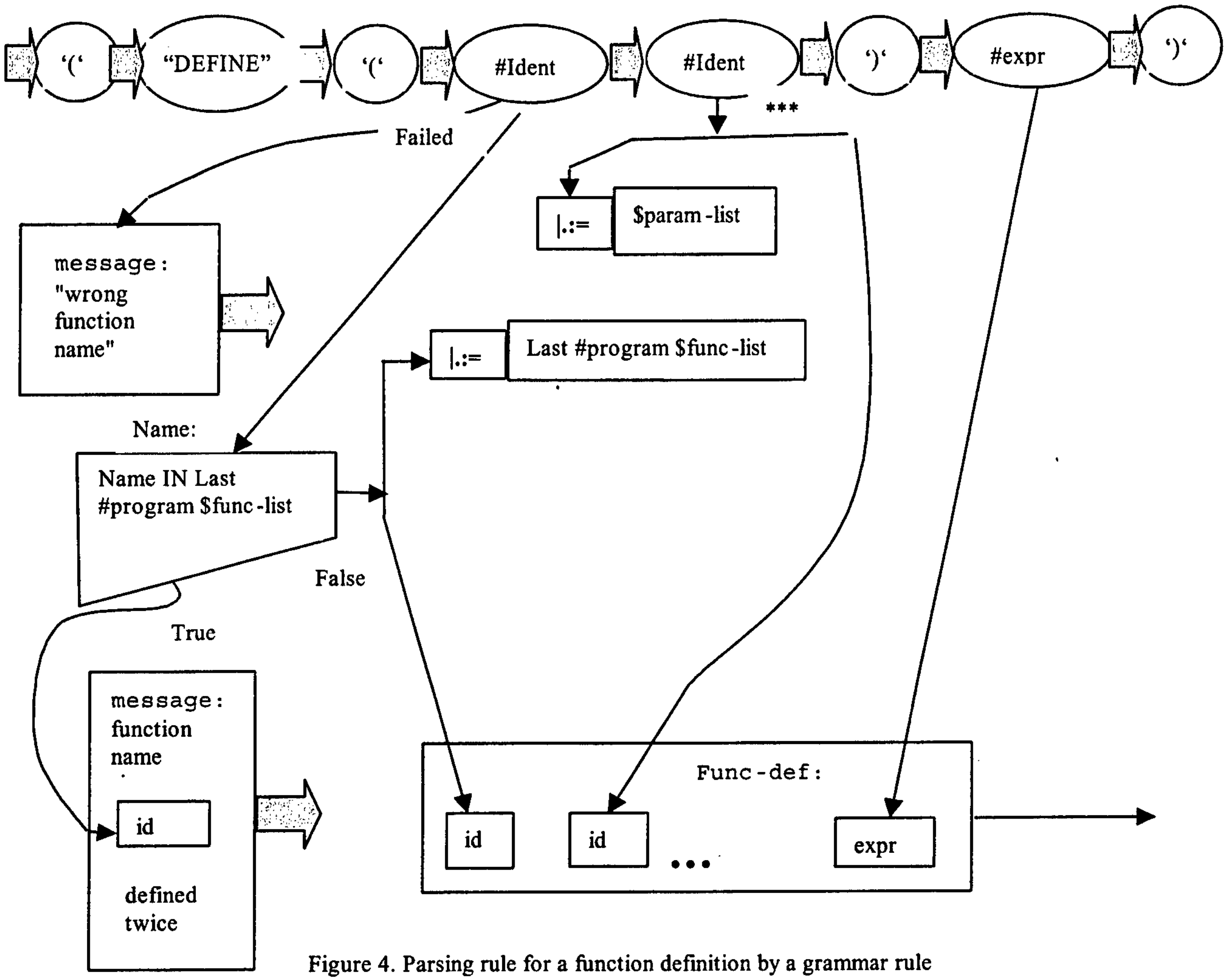


Figure 4. Parsing rule for a function definition by a grammar rule
Function-definition ::= '(' DEFINE '(' Name Param * ')' Expression ')'

Annotations for the rule #func-def

- Built-in rule #Ident matches a character string that is an identifier. When successful, this identifier (an object of the type id) is input to the conditional data flow switch to check whether the function name is already on the list. If true, the id item is forwarded to the message output stream. If false, it goes to the resulting tuple constructor.
- A function name is also sent to update state $func-list in the current instance of rule #program. |.:= stands for the operation to append an element to the end of list. This assignment operation updates the state Last #program $func -list.
- The entire sequence of patterns in this rule consumes part of the input stream delegated from the calling rule #program.
- Parameter names are appended to the state variable $param -list. All state variables are initialized by Null, which stands for empty list in this case.

`#expr : Stream [char] -> expr, Stream [message]`

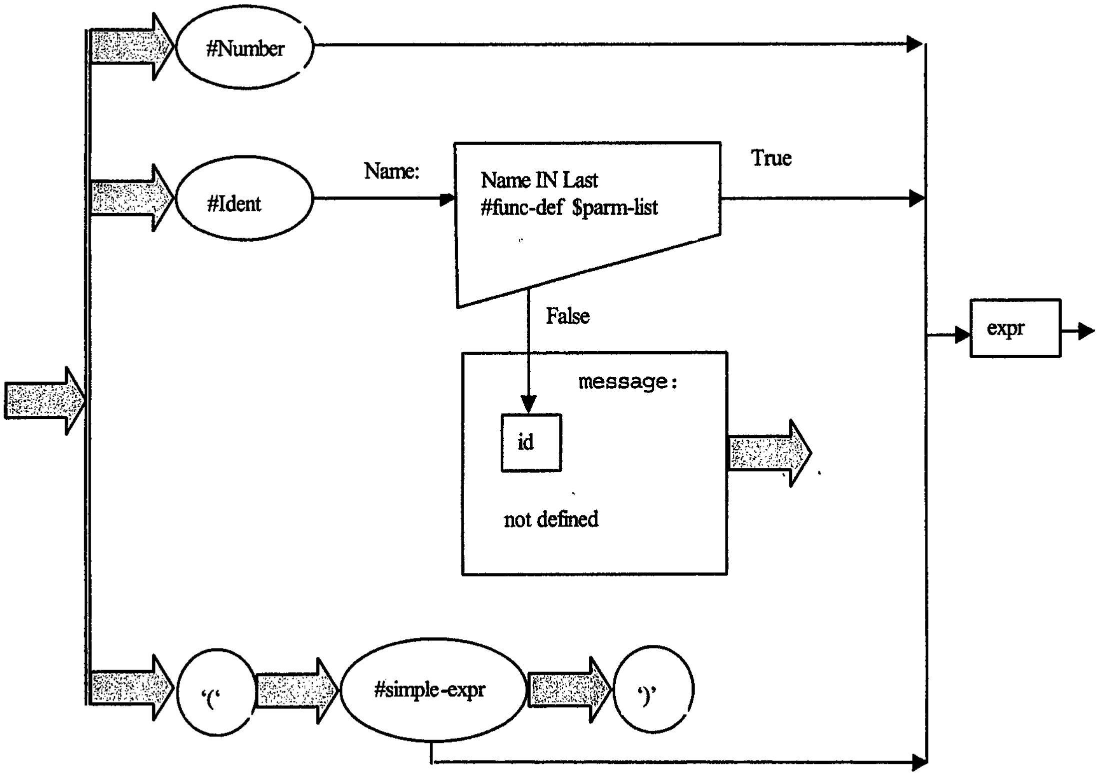


Figure 5. Parsing rule for MicroLisp e xpression for the grammar rule
`expression::= integer | parameter-name | '(' SimpleExpression ')'`

Annotations for the rule #expr

- A pattern may have several alternatives. The alternatives are applied in order of appearance, if the first alternative fails, the pattern matching backtracks in the input stream and the next alternative is applied until one of alternatives is successful. If all alternatives fail, the entire alternative pattern also fails.
- The built-in rules #Number and #Ident, when successful, return objects of the types `int` and `id`, correspondingly. Since the type expr is defined as a supert ype for `int` and `id`, the data flow to the resulting object of the type expr is consistent.

3.2 Code generation

Code generation rules take as input a MicroLisp abstract syntax object and output C abstract syntax objects. Target code template representation in th e diagrams is based on default mappings for C abstract and concrete syntax and visual repr e-sentation of append operation as nested boxes.

Annotations for the rule #gen-program

```
#gen-program: prog -> C-HeaderFile, C-CodeFile
```

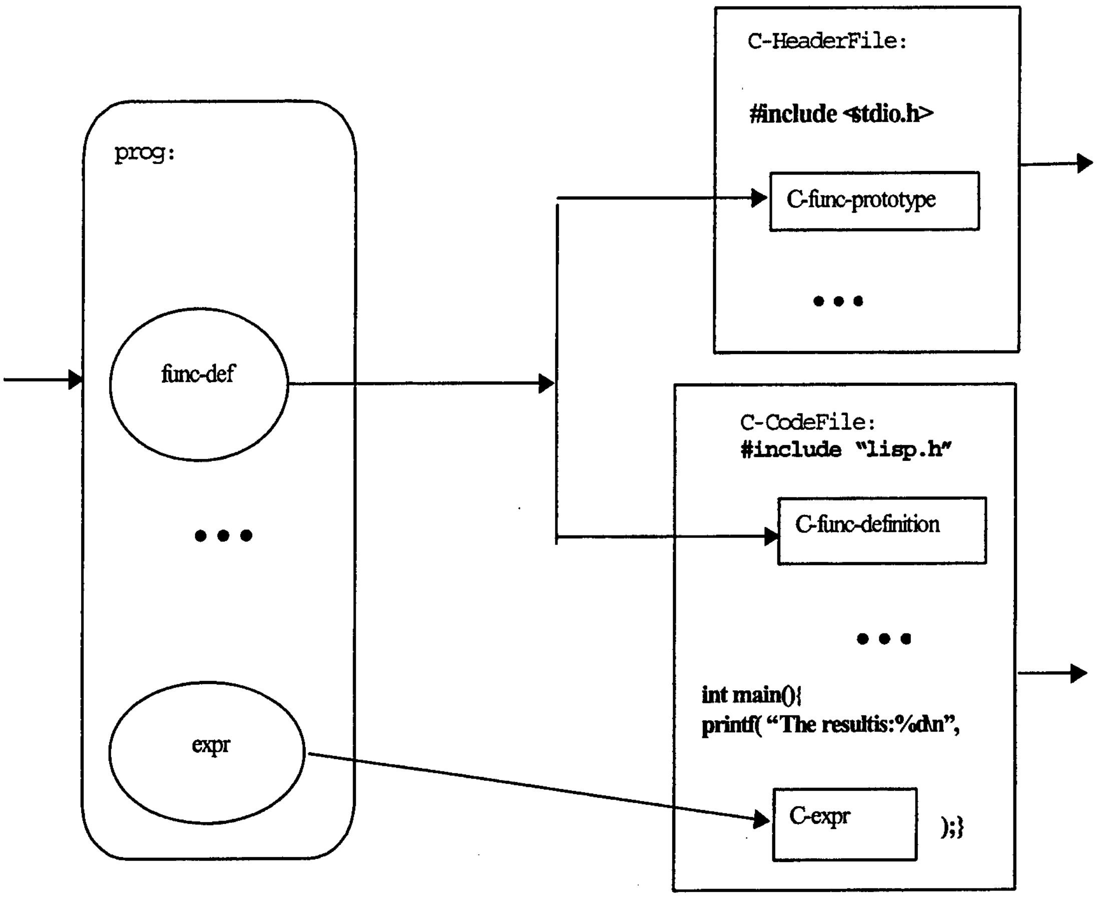


Figure 6. Generation rule for the MicroLisp program level

- The input is of the type prog (abstract syntax object for MicroLisp) and a pattern for this object provides an access to the component retrieval. Since func-def components may be repeated zero or more times, the ellipsis in the pattern represents the iterative traversal.
- The iteration of the input is synchronized with the iterative generation of objects in two outputs. The transformations itself are carried by default mappings `func-def -> C-func-prototype` and `func-def -> C-func-definition`. The rule #gen-function-prototype in the next example gives the algorithm for the first of these default mappings. Since the template provides particular concrete syntax for parts of the C code, those text segments will be stored with corresponding C abstract syntax objects. The resulting parse tree for include and printf will contain objects of the type id and text-string that hold values, such as "int", "printf", and other. These concrete syntax values are retrieved by default mappings when pretty-printing corresponding C abstract objects.
- The rule #gen-program constructs the target C code in the abstract syntax form. The mapping from abstract syntax to the text will be done according to the main diagram in Figure 1 by corresponding de-parsing default mappings for the C language. Both the abstract syntax definitions and default parsing and de-parsing mappings for the C language may be reused for any other meta-program that uses C as a target.

Annotations for the rule #gen-function-prototype

- This rule provides the flavor of hierarchical structure of generation te mplates.
- The first appearance of the str ing "int" in the target object C -func-prototype object will be converted by the C default parsing mapping into object C -type and the string "int" will be associated with it as a value. The same is true also for the iteration of "int" in the parameter list.
- Box around the second instance of "int" is needed to indicate the binding with the iteration of id in the source object func-def.

```
#gen-function-prototype: func-def -> C-func-prototype
```

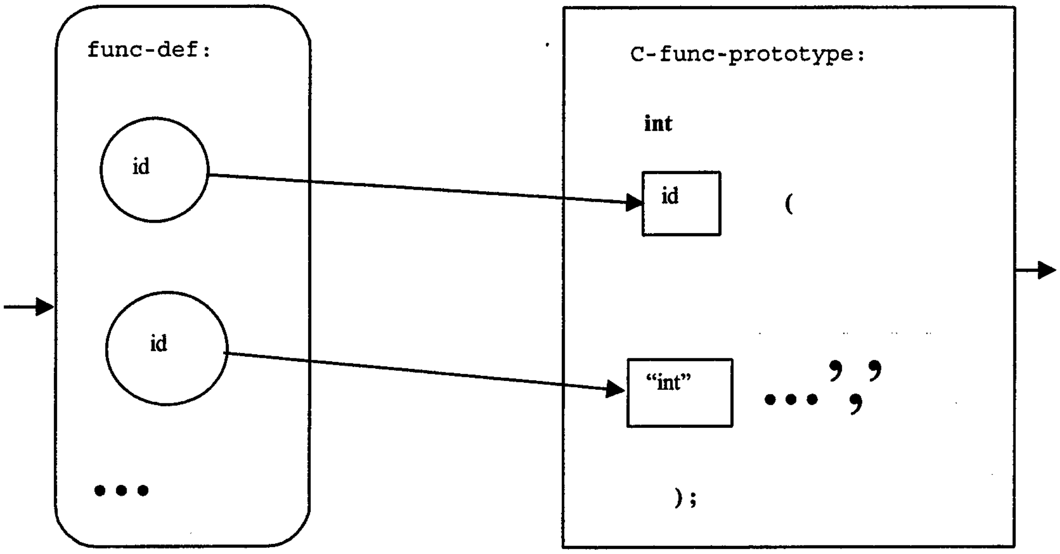


Figure 7. Generation rule for C function prototype.

- Parentheses, semicolon, and comma (as a separator between iterated elements; in the graphical interface there should be a way to indicate that comma is related to the iteration ellipsis, e.g. by a dashed box) in the target object are optional, and if present, will be consumed by corresponding C default parsing mappings. The resulting object is still an abstract syntax object.

4 Preliminary conclusions

This paper presents very preliminary results on the visual notation for meta -programming. Work continues on the la nguage itself, case studies, and implementation issues. At the moment of this writing the interpreter for the core of da ta-flow language is already implemented, and work is in progress on the graphical editor and advanced features like default ma ppings and tuple pattern matching. In it current form, the concepts presented may be used as a useful supplement to the meta-program design documentation. We expect the advantages of this approach to be as fo llows.

- Visualization of data and data flow provides for better readability and uncovers parallelism in data processing.
- The tuple type provides for a precise, disciplined, and f lexible way to define abstract syntax.
- The simple association mechanism provides a natural way to introduce data attributes and opens the road for pro cessing of arbitrary graphs without cluttering the language with additional means.

- Pattern matching notation covers in a uniform way data objects, rule calls, associations, and extended BNF notation for parsing.
- The language provides for systematic and consistent correspondence between constructors and patterns.
- The dynamic attributes (states) are actually macro extensions of pure functional paradigm (may be considered as additional inputs and outputs for diagrams referring to the states), provide for more efficiency, and make the data flow diagram simpler and less cluttered.
- Default mappings may be very convenient for generation templates, provide basis for lightweight type inference, and rule reuse.
- Data streams and patterns give a flexible and expressive framework for parsing rules supporting extended BNF notation, support reasonable and informative parsing error messages.
- Control mechanism, such as data flow switch, iteration and recursion fit well with data-flow notation and provide for transparent and expressive language to define different kinds of meta-programming algorithms.

References

[1] A.Aho, R.Sethi, J.Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986

[2] M.Auguston, "RIGAL - a programming language for compiler writing", Lecture Notes in Computer Science, Springer Verlag, vol.502, 1991, pp.529-564.

[3] M.Auguston, "Programming language RIGAL as a compiler writing tool", ACM SIGPLAN Notices, December 1990, vol.25, #12, pp.61-69

[4] M.Auguston, A.Delgado, Iterative Constructs in the Visual Data Flow Language, in Proceedings of IEEE Symposium on Visual Languages, Capri, Italy, 1997, pp.152-159

[5] E.Baroth, C.Hartsough, Visual Programming in the Real World, in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.21-42

[6] D.Batory, Gang Chen, E.Robertson, Tao Wang, Design Wizards and Visual programming Environments for GenVoca Generators, IEEE Transactions on Software Engineering, Vol. 26, No 5, May 2000, pp.441-452

[7] R. Bird, T. Scruggs, M. Mastropieri ,Introduction to Functional Programming, Prentice Hall, 1998

[8] P.T.Cox, F.R.Gilles, T. Pietrzykowski, "Prograph", in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.45-66

[9] K.Czarnecki, U.Eisenecker, Generative Programming, Methods, Tools, and Applications, Addison Wesley, 2000, pp.832, ISBN 0-201-30977-7

[10] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System, Communications of the ACM, 35(2):121-131, February 1992.

[11] J. Levine, T.Mason & D.Brown, lex & yacc, 2nd Edition, O'Reilly, 1992

[12] Reasoning Systems, "Refine User's Guide", Palo Alto, 1992

[13] The Vienna Development Method: The Meta-Language, D. Bjorner et al, eds, LNCS 61, Springer 1978

Appendix A. Syntax of MicroLisp language and an example of a program

```
Program ::= Function-definition* '?' Goal-Expression

Goal-Expression ::= Expression

Function-definition ::= '('DEFINE'('Function-name Parameter-name*')' Expression ')'

Expression ::= Integer | Parameter-name |'(' SimpleExpression ')'

SimpleExpression ::= BinOperation Expression Expression | UnOperation Expression |
                     Function-name  Expression* |COND Branch + | READ_NUMBER
```

```
Branch::= '('Expression Expression ')'
BinOperation ::= ADD | SUB | MULT | DIV | MOD | EQ | LT |GT | AND | OR
UnOperation ::=  MINUS |  NOT
Function-name  ::= Identifier
Parameter-name   ::= Identifier
```

Example of a MicroLISP program.

```
( DEFINE ( gcd  x y)
   (COND (EQ x y)   x )
 ( (GT x y) ( gcd  (SUB x y)   y ) )
 (   1 ( gcd  x (SUB y x)   ) ) )   )
?  (gcd  (READ_NUMBER)  (READ_NUMBER) )
```

Appendix B. Type definitions for MicroLisp -> C compiler

```
message::  [ char ]
program:: ( func_def* expr)| NULL
        attribute func_tab: [id]
func_def:: id id* expr
expr:: number | id |(op expr expr)|(op expr)|read_num | cond | function -call
function-call:: id  expr*
cond:: (expr expr)*
default mappings
 #prog:  [ char ] -> prog
 #gen_program: prog -> C-HeaderFile, C-CodeFile
 #gen-function-prototype: Func-def -> C-func-prototype
 #gen-function-def: Func-def -> C-func-definition
 #pretty_print_prog: prog -> [ char ]
 ........
```

This is a sketch of a (over)simplified version of C abstract syntax.

```
 C_CodeFile::  include-statement * C-func-definition +
 C_HeaderFile:: include-statement C-func-prototype *
 C_func_prototype:: C-type func-name C-type *
 C-type:: id
 C_func_definition:: .........
 C_expr:: .....
```

Default mappings include parsing rules and pretty -printing rules (abstract syntax to text mappings).

Optimization of Distributed Object-Oriented Servers

William J. Ray
SPAWAR Systems Center, (619) 553-4150, ray@spawar.navy.mil

Valdis Berzins
Naval Postgraduate School, (831) 656-2610, berzins@cs.nps.navy.mil

Abstract - This paper presents a method for deploying distributed object servers to optimize client response time. Object-Oriented (OO) computing is fast becoming the de-facto standard for software development. Distributed OO systems can consist of multiple object servers and client applications on a network of computers, as opposed to a single large centralized object server. Optimal deployment strategies for object servers change due to modifications in object servers, client applications, operational missions and changes in various other aspects of the environment.

As multiple distributed object servers replace large centralized servers, there is a growing need to optimize the deployment of object servers to best serve the end user's changing needs. A method that automatically generates object server deployment strategies would allow users to take full advantage of their network of computers.

States of the art load balancing techniques schedule a given number of independent tasks on a set of machines. However, object servers do not have independent tasks: all methods in an object are related. Also, the number of times a method is called is usually dependent on interactions with end users.

The proposed method profiles object servers, client applications, user inputs and network resources. These profiles determine a system of non-linear equations that is solved to produce an optimal deployment strategy.

Keywords: Distributed Object, Load Balancing, Client Response Time, Optimization, Server Deployment and Software Engineering.

1. INTRODUCTION

The future of computing is heading for a universe of distributed object servers. The evolution of object servers to distributed object servers will parallel the evolution of the relational databases. Over time, object servers will provide functionality to more client applications than their original applications, just as relational databases were used by more applications than the original application. In both cases, systems optimized for the original application may not perform well for the new applications. Tools that allow a programmer to model an object and easily create object servers with all the necessary infrastructure code needed to work as a distributed object server are available [12]. This will lead to an explosion in the number of object servers available to client applications.

A user's network of computers will change frequently. Object servers, applications, hardware and user preferences will be in a constant state of flux. No static deployment strategy can adequately take advantage of the assets accessible on the network in such an environment.

No system can accurately predict user interaction with a system. Two separate users performing the same job will interact with a system differently. The same user may interact differently while performing the same job at different times. For these reasons and combinatorial explosion problems, an adaptive software engineering approach is proposed instead of a traditional computer science approach.

Most deployment strategies today are dictated by the system engineer's view of how the systems will be utilized. Of course, the system engineer doesn't revisit these strategies every time hardware, software or user interactions change. The goal is to allow the user to update hardware and usage profiles. Software developers would supply new profiles when their code changes. Any time a profile is updated, the model would be run and an automated reconfiguration of the object server deployment could occur. In most cases, the frequency of change will be greatest in the hardware and usage pattern profiles. Since many of these changes can take place without the knowledge of a system engineer or the budget to employ one, a method that allows the users to update these profiles and initiate the reconfiguration is desired.

2. PREVIOUS WORK

There has been little work on deployment strategies for distributed object servers. The closest relevant research is in the fields of load balancing, client/server performance and distributed computing. Most state of the art load balancing techniques address scheduling of given set of tasks on a set of given machines. Some techniques only deal with tasks that are independent. Others deal with dependent tasks that are usually linked together by temporal logic and mutual exclusion constraints.

Object servers do not have independent tasks. All methods in all object types in a single object server are related at least by locality and more often by the interaction between the object types. Also, the number of times a method is called is not given, but rather depends on undetermined interactions with end users, very much like the situation in client/server performance research. We propose a system that enables optimization of object server deployment to meet changing needs.

3. CURRENT PRACTICES

Because of the difficulty in producing the infrastructure code necessary to support distributed object computing, many developers produce huge monolithic object servers [11]. A powerful machine is usually needed to adequately handle this server and successful applications that experience large increases in the number of users may outgrow the capabilities of the fastest available single machine. With automated code-generation tools, these servers will be much easier to produce and reconfigure [12]. This allows servers to be partitioned by allocating unrelated or loosely related objects types to different physical servers that can be deployed across the network to take advantage of the available assets. By taking advantage of all the assets on the network, faster response times can be achieved [11].

Loosely related object types are defined as object types that contain associations to other object types. When these object types reside in different physical object servers, the result is an object server that calls on other object servers. A server that calls other servers is a complex server [1].

Many networks of computers are installed with a single purpose in mind. Over time, these networks support an evolving set of tasks. Even though the original role the network played can change dramatically, rarely does a single system engineer revisit the deployment strategy for the entire system. What a user ends up with is usually the product of multiple system engineers' choices made based on the latest incremental changes without regard for the system as a whole and interactions among its roles. It is infeasible, because of cost, to hire a system engineer to re-assess the whole system every time a change occurs. In the end, the user is left with a system whose deployment strategy borders on randomness.

4. OPTIMIZATION OF DISTRIBUTED OBJECT-ORIENTED SYSTEMS

The goal of this paper is to describe a method that can generate distributed object oriented server deployment architectures to take advantage of network resources for the purpose of reducing average client response time. A system that carries out this method must be able to reason about deployment strategies of loosely related objects. The proposed system maps all of these profiles into equations to minimize average client response time.

Average client response time was chosen as the optimization criteria over others. In this paper, the goal was to be user centric. Criteria that focused on maximizing machine utility were not germane. Average client response time was chosen over minimizing the maximum response time of one call because the method takes into account the entire usage profile.

4.1 Optimization Model

The equations that need to be solved will minimize the sum of all of the response times for a given call pattern over a given time interval. Since we want to allow the user the freedom to run client applications from anywhere on the network, we will ignore all processing on the client machines and all network delay between client machines and server machines. The only factors we will consider for optimizing our server deployment are the processing on the object server and the network delay between complex object servers. Therefore, the objective function that we wish to minimize is:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * R_n * S_{norm}}{S_m} + \sum_{i=0}^{N} \sum_{j=0}^{N} \frac{B_{ij}}{Q_{ij}} \right]$$

subject to the following four constraints:

1. Object Servers cannot be split across machines.

a_{nm} = 1, iff server n is running on machine m
0, otherwise

2. Each Server can run on only one machine [no multiple instances of the same server.

$$\forall n \left[\sum_{m=0}^{M} a_{nm} \equiv 1 \right]$$

3. RAM usage by the object servers cannot pass a set threshold on each machine.

$$\forall m \left[\sum_{n=0}^{N} a_{nm} * V_n \leq T_m * U \right]$$

4. CPU time on a given machine cannot surpass the corresponding real time interval.

$$\forall m \left[\sum_{n=0}^{N} \frac{a_{nm} * R_n * S_{norm}}{S_m} \leq C \right]$$

where

N	= Number of object servers
M	= Number of physical machines
a_{nm}	= server n is running on machine m
R_n	= Normalized machine load of server n (seconds, s)
S_{norm}	= Speed of the normalizing machine (MHz)
S_m	= Speed of machine m (MHz)
B_{ij}	= Data sent between server i to server j (bits, b)
Q_{ij}	= Network Speed between server i to server j (bps)
T_m	= Physical RAM on machine m (bits, b)
V_n	= Memory allocated by server n (bits, b)
U	= Multiple to limit RAM utilization [0.1,3.0]
C	= Time Interval [seconds, s]

NOTE: All terms are fixed either by measurement or input except for a_{nm}. The model varies all possible combinations for a_{nm} and finds the minimum based on the above objective function and constraints.

4.2 Evolution

Over time, a collection of hardware, software and user requirements will change in a given environment. Common hardware changes consist of adding new computers, removing old computers, upgrading CPUs, modifying RAM and modifying network bandwidth capacity. Each of these hardware changes will produce an event that would trigger the system to re-evaluate its deployment strategy.

Software can also be quite dynamic in nature. New object servers and applications can appear. Old ones can be removed. Existing object schemata and methods can be changed. Each of these changes would trigger an event to re-evaluate the deployment strategy.

4.3 Loosely Related Objects

Not all objects types that are related must necessarily be contained in a single object server. There is a point where the performance of the system would improve by moving the object type into a different server. This is usually the case when none of the application code exercises an inter-server method call or exercises it only very rarely. Large message sizes and slow network speeds will push for related object types to be co-located. The approach will be able to reason about not only deploying object servers, but also recommend the schema supported by these object servers.

4.4 Priority Setting

User requirements can also be in a state of flux. Most computer systems are used to support multiple jobs. Business-hour requirements can differ greatly from after-hours computational requirements. A developer's network of computers can support multiple projects, but may need to be optimized for a single project for demonstrations. In the military, the operational mission being supported can change significantly. For example, a set of distributed object servers could be used to support many applications aboard a ship. These applications could handle such tasks as Anti-Submarine Warfare (ASW), Anti-Surface Warfare (ASUW), Anti-Air Warfare (AAW), Electronic Warfare (EW), humanitarian missions and rescue missions. The relative computational activity of these applications could differ significantly on different missions of the ship.

Optimizing a system of object servers for all possible roles would not be optimal when the system is only performing a couple of missions at a time. By profiling each role, the user could choose to re-optimize his deployment to decrease the response time when user chosen roles change. In this way, the user could tune his system to give peak performance for the task he is currently trying to perform.

4.5 Profiles

The tricky part is to figure out what elements are needed in the different profiles, how to map these profiles into equations and then model how these profiles interact with each other. The more complex the modeling of the hardware becomes the more computationally intensive the approach will become. Initially we demonstrate an approach with rather simplistic profiles to demonstrate its capabilities.

4.5.1 Hardware Profiles

The aspects being modeled in the hardware profiles include characteristics of each computer such as CPU speed and physical RAM size. The hardware profile also models the network speed between each computer. Current hardware profiles do not directly support multi-processor computers, but they could be modeled as groups of separate nodes with very high "network speeds" between them.

4.5.2 Object Server Profiles

Object servers need to be profiled for metrics associated with each method call in each object. The computational time of each method call should be captured and normalized to a specific hardware architecture. Since object servers ideally run continuously, the RAM of the object server must also be measured and summarized. The hardware profile and the object server profile is sufficient to optimize the server deployment for the case where all the functionality contained in all the objects is of equal value to the user. Metrics can be collected easily with a small client application that exercises each method call and records the data. Thus, actual implementation code for the application isn't needed to estimate the object server profiles.

4.5.3 Client Application Profiles

Ideally, client applications would be delivered with their profiles. If the code is available, then the source can be parsed to find all possible object invocations. Since exact frequencies of method calls are not algorithmically computable in the general case, measurement is necessary to reliably estimate frequencies of calls. The system must allow a user to create typical scenarios and record the method calls that occur in the scenario. This could be done by simulation or monitoring calls to the object servers when the system is in a training mode. The plus side to this method is that the user could represent more complex tasks involving many user interactions in a single profile. Numerous tools exist for complex event processing in a distributed system [5, 6].

4.5.4 User Profiles

User profiles or roles indicate how a user interacts with the system over a given period of time. In simplistic terms, it is like keeping track of how many times each button is selected over a given time interval. Average button push rates can be expressed as number of events per second. The user can collect this data manually or automatically by the system with audit trails. Multiple roles can exist for each user. The user could then select a set of roles and have the system come up with an optimal deployment strategy to meet these criteria.

4.6 Profile Mappings

In order to compute the optimal deployment strategy given a set of profiles, one needs to map these profiles into equations that can be solved for minimum response time. To illustrate the mappings, we present an example. The example consists of three machines, three object servers and three client applications. The method demonstrates the differences in deployment for a system tuned to a users-specific role. Table 1 shows the profile for the computer hardware available.

Table 1. Machine profile for example.

MACHINE	RAM (bits)	CPU Speed (MHz)
SIX	512,000,000 = 64MB	600
BR733	1,024,000,000 = 128MB	733
GIGA	1,024,000,000 = 128MB	1000

Table 2 shows the network bandwidth available to communicate from each machine to the other. In this example, the machines will have equal bandwidth between machines as is the case when all servers are running on the same local LAN. The speed of communications between servers on the same machine is more difficult to predict. These speeds usually lie in the interval bounded by the speed of the machines back plane and the speed of the network. It is dependent on the operating system, implementation of the middleware, and other factors. For this example, we assume that intra-machine communication is twice as fast as inter-machine communication. In the absence of measurements, the system can be run with best and worst case scenarios by specifying the boundary values identified above.

Table 2. Network speed.

Machine to Machine Speed (bps)	SIX	BR733	GIGA
SIX	200,000,000	100,000,000	100,000,000
BR733	100,000,000	200,000,000	100,000,000
GIGA	100,000,000	100,000,000	200,000,000

Besides the hardware profiles, we need to have the server profiles. Table three lists each server's RAM requirements.

Table 3. Server RAM requirements.

SERVER	RAM Required (bits)
A	352,000,000 = 44MB
B	480,000,000 = 60MB
C	528,000,000 = 66MB

Additional parts of the object server are the timing of each individual method call available in each server and a list of complex method calls. All of these measurements were taken on a single machine to normalize the values. In this example, server A has one four methods, server B has two methods, and server C has three methods.

Table 4. Normalized Server Loads.

SERVER	Method	CPU time (s)	Average Size of Message (b)
A	1	0.5796	112000
A	2	2.6203	18400
A	3	1.18175	44800
A	4	2.0264	176000
B	1	1.76655	4000000
B	2	3.70085	2720000

C	1	3.0043	320000
C	2	4.8040	4000000
C	3	0.48815	400000

A complex method call is a method call that calls another object server. These method calls require special handling in measuring their load on the host server and in the objective function for optimizing the system. Table 5 lists the complex method calls in this example.

Table 5: Complex Method Calls

Complex Method	Exterior Calls
B.2	C.1

The last information needed to optimize the system is information about the applications and the users. This step adds roles to the list of profiles for the system to optimize. These roles have more realistic use patterns for the different jobs a user would actually perform on the system. For this example, we will have three client applications with two buttons, nine buttons and three buttons respectively.

Let's assume that there are three different roles the network of computers supports for the user and the following is the use pattern shown in Table 6, and that the buttons call the following server methods shown in Table 7. Method calls that appear in italics in Tables 7 and 8 are complex method calls. They appear in italics to remind us that these methods require special handling when figuring out the objective function.

Table 6. Roles.

ROLE	CALL PATTERN (observation interval is 990 seconds)
Role 1	50 C1.B1 + 1 C1.B2 + 1 C2.B1 + 1 C2.B6
Role 2	10 C1.B1 + 40 C1.B2 + 24 C3.B2
Role 3	50 C2.B5 + 10 C2.B9 + 30 C2.B3 + 1 C2.B2 + 1 C3.B2

Table 7. User interface calls.

Button	Methods Called
C1.B1	A.1
C1.B2	A.2 + B.1
C2.B1	C.1 + C.2
C2.B2	C.3
C2.B3	C.2
C2.B4	C.3
C2.B5	A.1 + _B_.2
C2.B6	_B_.2
C2.B7	A.4
C2.B8	C.3 + A.3
C2.B9	A.1 + A.2 + A.3 + _B_.2
C3.B1	C.1
C3.B2	B.1 + _B_.2
C3.B3	C.2

By substituting the user interface calls into the roles matrix, we get an objective function for optimizing the system shown in Table 8. All other method calls will be ignored.

Table 8. Roles to server calls.

ROLE	Methods Called in Role
Role 1	50 * (A.1) + 1 * (A.2 + B.1) + 1 * (C.1 + C.2) + 1 * (_B_.2)
Role 2	10 * (A.1) + 40 * (A.2 + B.1) + 24 * (B.1 + _B_.2)
Role 3	50 * (A.1 + _B_.2) + 10 * (A.1 + A.2 + A.3 + _B_.2) + 30 * (C.2) + 1 * (C.3) + 1 * (B.1 + _B_.2)

4.6.1 Filling in the Equation for Role 1

Role 1 consists of 50 C1.B1 calls, one C1.B2 call, one C2.B1 call, and one C2.B6 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 4.

50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [*B.2*] =
50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [B.2 + C.1] =
50 A.1 + A.2 + B.1 + C.1 + C.2 + B.2 + C.1 =
50 A.1 + A.2 + B.1 + B.2 + 2 C.1 + C.2

This leads to the following values for the array R for the optimization equation.

R (A) = 50 [A.1 values for CPU] + 1 [A.2 value for CPU]
= 50 [579.6] + 1 [2620.3]
= 31600.3

R (B) = 1 [B.1 values for CPU] + 1 [B.2 value for CPU]
= 1 [1766.55] + 1 [3700.85]
= 5467.4

R (C) = 2 [C.1 values for CPU] + 1 [C.2 value for CPU]
= 2 [3004.3] + 1 [4804.0]
= 10812.6

There is only one italicized method call prior to substitution, so there is only one network value to deal with.

BITS[B,C] = 1 [*B.2* message in bits]
= 320000

4.6.2 Filling in the Equation for Role 2

Using the same approach as in 4.6.1, we get the following for Role 2:

R (A) = 110608
R (B) = 201879.6
R (C) = 72103.2

There is only one italicized method call prior to substitution, so there is only one network value to deal with. However, it is called 24 times.

BITS[B,C] = 24 [*B.2* message in bits]
= 24 [320000]
= 7680000

4.6.3 Filling in the Equation for Role 3

R (A) = 72796.5
R (B) = 227518.4
R (C) = 327870.45
BITS[B,C] = 19520000

4.7 Model Solutions

All of the information above is run through a LINGO model that varies the location of the object servers on the different machines to find the a solution set that minimizes the value of the objective function. The model prompts the user for inputs bandwidth, RAM percentage and computational time limitations. Changing any of these variables will lead to different model outputs [10].

4.8 Model outputs

This method outputs the following deployment strategies for the different roles when setting different RAM limits and keeping all other variables the same as in the last example. Solving the optimization problem defined in section 4.1 with the parameter values determined in section 4.6 derives these results.

Table 9. Single user deployment strategies for different roles. RAM limit set to 1.5.

Machine	Role 1 (user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	None	None	None
GIGA	A, B, C	A, B, C	A, B, C

Table 10. Single user deployment strategies for different roles. RAM limit set to 1.0.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	B	C	A
GIGA	A, C	A, B	B, C

Table 11. Multiple concurrent users deployment strategies for different roles. RAM limit set to 1.0.

Machine	Role 1 (28 user)	Role 2 (4 user)	Role 3 (3 user)
SIX	None	A	A
BR733	B, C	C	B
GIGA	A	B	C

From the model output, we can see that when a single user is present and RAM is not a limiting factor, the result is that all the servers migrate to the fastest machine. However, when we start to limit RAM, the servers start to spread out. The first server to leave the fastest machine turns out to be different in each role. Multiple concurrent users also tend to spread the servers across the available machines. The significance of the model is that different roles and different numbers of concurrent users lead to different optimal configurations in most cases for this example. No single static configuration can outperform the ability to change configurations based on perceived changes in the usage of the system.

4.8 Experimentation

We tested the validity of the model by experimental measurement. A testbed was created with Windows 2000 machines that match the characteristics of the machines in the above example. Servers were created using JDK 1.3 and RMI as the middleware. Software to simulate the three different users was also created. The user was simulated with a random choice for button selection that has a uniform distribution similar to the roles. This simulation software was instrumented to measure the actual time the software was blocked waiting for an object server method call to response [10]. All 27 different configurations were established and the average response time for each configuration was measured and recorded. Between each simulation, the testbed machines were rebooted.

All 27 configurations were tested twice. One tested the configuration with the object servers using much less than the stated memory needs. Another tested the configuration with the object servers using all of the stated memory needs. Some configurations strained the machines memory limits. These configurations resulted in system failures in the test with the object servers using all of the stated memory needs. These system failures are listed as error in the tables of results. It should be noted that Windows 2000 did a much better job of swapping when memory utilization exceeded 100% than a previously tested operating system, Windows NT.

4.8.1 Experimentation Results

The below table is a tabulation of experimental results obtained from measuring the outputs of a test system.

Table 12: Measured Response Times

PAT	A	B	C	ROLE 1	ROLE 2	ROLE 3	R1 MEM	R2 MEM	R3 MEM
1	GIGA	GIGA	GIGA	976.331	5150.362	6741.948	977.343	5120.184	6776.846
2	GIGA	GIGA	BR733	899.344	5530.329	8266.516	942.984	5580.438	8213.157
3	GIGA	BR733	GIGA	960.811	6417.171	7802.172	887.031	6349.859	7900.562
4	GIGA	BR733	BR733	1079.641	6686.376	9124.938	1041.391	6696.141	9217.953
5	BR733	GIGA	GIGA	1140.796	5953.015	7413.343	1144.672	5874.642	7267.639
6	BR733	GIGA	BR733	1218.875	6233.064	8508.343	1282.643	6204.922	8519.844
7	BR733	BR733	GIGA	1119.092	6877.968	8142.719	1228.031	6838.001	8232.064
8	BR733	BR733	BR733	1186.861	7238.876	9428.658	1409.515	7215.576	9373.861
9	GIGA	GIGA	SIX	991.531	5958.547	9259.221	1039.298	5916.187	9463.079
10	GIGA	SIX	GIGA	878.782	7176.861	8627.407	962.609	7288.954	8532.983
11	GIGA	SIX	SIX	1157.765	7852.795	10712.984	error	error	error

12	SIX	GIGA	GIGA	1274.376	6375.549	7332.718	1348.828	6424.484	7346.219
13	SIX	GIGA	SIX	1402.687	6969.187	9838.221	error	error	error
14	SIX	SIX	GIGA	1413.983	8211.857	8972.002	error	error	error
15	SIX	SIX	SIX	1642.232	8644.362	12131.091	error	error	error
16	BR733	BR733	SIX	1197.423	7342.092	10387.125	1262.703	7322.595	10529.611
17	BR733	SIX	BR733	1306.374	7862.331	10360.985	1439.251	8148.969	10123.563
18	BR733	SIX	SIX	1305.296	8514.078	11067.388	error	error	error
19	SIX	BR733	BR733	1291.719	7601.829	9591.424	1535.657	7742.921	9770.578
20	SIX	BR733	SIX	1467.437	8033.173	10590.126	error	error	error
21	SIX	SIX	BR733	1441.421	8222.031	10185.453	error	error	error
22	GIGA	BR733	SIX	1114.344	6987.719	10259.391	982.687	6967.624	10193.641
23	GIGA	SIX	BR733	1068.765	7423.048	9834.875	1131.969	7343.782	9804.983
24	BR733	GIGA	SIX	1246.361	6515.812	9563.001	1311.905	6613.031	9617.297
25	BR733	SIX	GIGA	1304.703	7783.171	8743.235	1189.655	7548.561	8865.811
26	SIX	GIGA	BR733	1355.594	6752.499	8625.439	1390.297	6772.453	8860.094
27	SIX	BR733	GIGA	1306.687	7380.828	8259.047	1344.611	7457.968	8328.064

4.8.2 Role 1

The models chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the third fastest average response time in the minimal memory run and the fastest average response time in the stated memory run. The fact that pattern 10 was the fastest average response time in the minimal memory run is a result of the variability of the simulation [10]. Pattern 1 was the fourth fastest on both runs even though it was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. More interesting from a software engineering standpoint was the fact that the model proposed a configuration that outperformed most configurations from 10 to 44 percent and that the recommended patterns were free from failures.

4.8.4 Role 2

The models predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. In the two runs, the models predicted configuration of pattern 2 was the second fastest average response time in both runs. Pattern 1 was the fastest average response in both runs, which is the predicted configuration when RAM usage is 150% of physical RAM. Again, the configuration chosen by the model outperformed most configurations from 10 to 38 percent.

4.8.5 Role 3

The models predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. In the two runs, the models predicted configuration of pattern 5 was the third fastest average response time in the minimal memory run and the second fastest average response time in the stated memory run. Pattern 1, the fastest average response time in both runs, was the predicted configuration when RAM usage was set to 150% of physical RAM. The fact that pattern 12 was the second fastest time in the minimal memory run is a result of the variability of the simulation [10]. Again, the model proposed configuration outperformed most configurations from 10 to 44 percent.

5. CONCLUSION

The approach seems to have merit and produce useful results. The system responds in a reasonable way with changes is the environment, constraints placed on the system, and different roles that a user might want. Since all of these changes take place on a given network of computers, static deployment strategies will never utilize the assets available to better support the end user. The strategies chosen by our model were robust in the sense that performance was good even when actual loads departed from predicted loads.

Predicting exactly how a user will interact with a system that supports multiple roles will always be an inexact science. This system provides an adaptive software engineering approach to a real world problem that currently does not have a better solution. No solution can be exact because of the limitations inherent in modeling users, software, hardware, etc.

Perhaps the most significant capability added by our model is the ability to automatically grow to the point where machine limits are exceeded and hard failures occur.

6. FUTURE WORK

The system needs to be refined to more precisely reflect the workings of the network of computers. These refinements include allowances for asymmetric communications, more precise models for computers, operating systems, middleware, and queuing delays. Aggregated tuples of these models will be necessary to better evaluate the impact of RAM utility on processing speed.

Tools will also need to be produced to ease the collection of data for the profiles. The initial prototype uses a manual process involving LINGO 6 using data from previously collected metrics. The ability to easily collect the necessary metrics and automatically solve the problem is desirable. A tool that maintained roles and could start the servers on the given machines for that role would also be helpful. In a mature system, the tools should also automate the server code generation and reconfiguration processes.

The approach could also be used to optimize other kinds of systems involving servers, such as web sites and relational databases by modeling each server as an object. This would enable better deployment strategies, especially since many of these non-object servers could be tightly coupled to object servers. Of course, combinatorial explosion is also an issue. Larger systems can cause significant delays in computing deployment strategies. More realistic models as mentioned above could also significantly impact the processing time.

REFERENCES

[1] Adler, R., "Distributed Coordination Models for Client/Server Computing," *IEEE Transactions on Computers*, pp. 14-22, April 1995.

[2] Berzins, V. and Luqi, "Software Engineering with Abstractions", chapter 6, Addison-Wesley, ISBN 0-201-08004-4, 1991

[3] Kim, J., Lee, H. and Lee, S., "Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 499-505, April 1997.

[4] Loh, P., Hsu, W., Wentong, C. and Sriskanthan, N., "How Network Topology Affects Dynamic Load Balancing," *IEEE Transactions on Parallel and Distributed Technology*, vol. 4, no. 3, pp. 25-35, Fall 1996.

[5] Luckham, D. and Frasca, B., "Complex Event Processing in Distributed Systems," *Computer Systems Laboratory Technical Report CSL-TR-98-754.* Stanford University, Stanford, 1998.

[6] Luckham, D. and Vera, J., "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995.

[7] Lui, J., Muntz, R. and Towsley, D., "Bounding the Mean Response Time of the Minimum Expected Delay Routing Policy: An Algorithmic Approach," *IEEE Transactions on Computers.* Vol 44, No. 12, December 1995, pp. 1371-1382.

[8] Mehra, P. and Wah, B., "Synthetic Workload Generation for Load-Balancing Experiments," *IEEE Transactions on Parallel and Distributed Technology*, vol. 3, no. 3, pp. 4-19, Fall 1995.

[9] Perrochon, L., Mann, W., Kasriel, S. and Luckham, D., "Event Mining with Event Processing Networks," *The Third Pacific-Asia Conference on Knowledge Discovery and Data Mining*. April 26-28, 1999. Beijing, China, 5 pages.

[10] Ray, W., "Optimization of Distributed, Object-Oriented Systems," *PhD Dissertation in Software Engineering*, Naval Postgraduate School, September 2001.

[11] Ray, W., Berzins, V. and Luqi, "Adaptive Distributed Object Architectures," *AFCEA Federal Database Colloquium 2000 Proceedings*, pp. 313-330, September 2000.

[12] Ray, W. and Farrar, A., "Object Model Driven Code Generation for the Enterprise," *IEEE RSP 2001*, June 2001.

Formalizing Software Architectures for Embedded Systems

Pam Binns and Steve Vestal
pam_binns@htc.honeywell.com   steve_vestal@htc.honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

This paper outlines an approach to embedded computer system development that is based on integrated use of multiple domain-specific languages; on increased use of mathematical analysis methods; and on increased integration between domain-specific specification and mathematical modeling and code generation. We first outline some general principles of this approach. We then present a bit more detail about the emerging SAE standard Avionics Architecture Description Language and our supporting MetaH toolset. We conclude with a summary of some research challenge problems, technical approaches, and preliminary results uncovered during our work.

1 Introduction

The use of domain-specific languages (4GLs) and tools for embedded applications is wide-spread and will increase. A number of COTS tools are already in wide use for the development of feed-back control and display applications, for example. In many cases the use of domain-specific technologies in preference to general-purpose software development technologies can result in cost savings and improvements in quality factors that justify the additional development and acquisition costs of the domain-specific tools. Meta-tool technologies are available and have been used for lower-cost development of specialized domain-specific tools[5].

Three main elements of a domain-specific language and toolset are illustrated in Figure 1. There is the domain-specific language and editor, which allows concise and rigorous specification of the structure and semantics relevant to a particular engineering discipline. There are modeling and analysis methods and tools to support design during the early phases of development and verification during the later phases of development. There is a code generation or synthesis method to produce an implementation from a design specification. We believe these elements should be integrated and automated as much as is practical. Models and code should be generated from a common specification, eliminating the hand-development of separate model specifications where possible. The mapping between specification, models and code should be structure-preserving, intuitive, and easily verified. It should be possible to easily trace in any direction between design specification, models, model analysis results, and code.

The construction of complex embedded computer systems is an inherently multi-disciplinary effort and requires integrated use of multiple domain-specific languages and tools. The mix of specification languages and tools needed in a particular development environment will depend on the mix of embedded functionality needed in a particular product line.

Our work has focused on applying the above principles in the development of a computer system architecture specification language and toolset. This language and toolset are designed for use by embedded computer system architects, among whose tasks is the integration of various hardware components and software applications developed by other engineering groups using other domain-specific languages and tools. Figure 2 illustrates how the outputs of multiple domain-specific tools feed into the architecture specification toolset, which supports computer system integration and modeling and analysis. The output of the toolset, in addition to models and analysis results, is software that integrates the pieces of the system together.

In addition to supporting embedded system developers, the language and toolset also provide a context and enabler for technology development activities. We are using the language and toolset as an object of study and a prototyping testbed in research activities intended to enable large, dynamically recon-

[*] This work has been supported by DARPA, Army AMCOM, AFOSR, and Honeywell Laboratories.

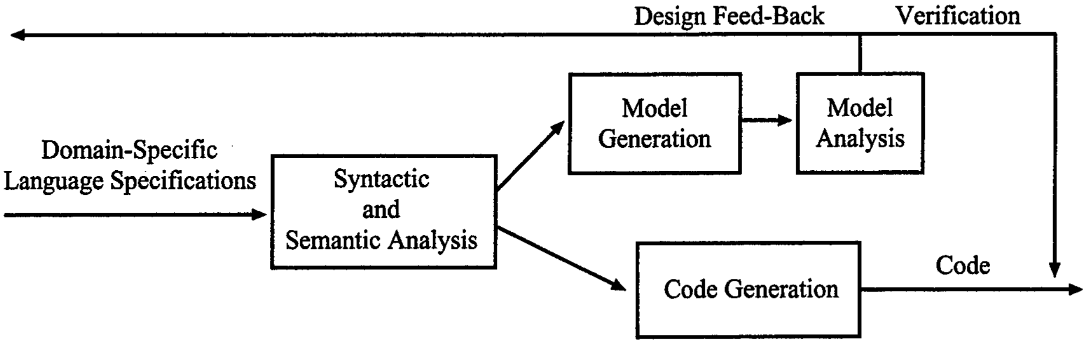


Figure 1: Notional Domain-Specific Toolset

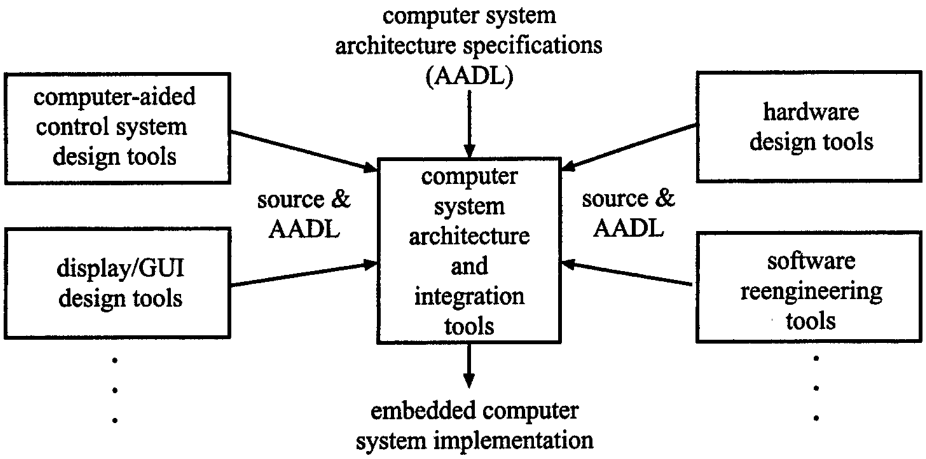

computer system architecture specifications (AADL)
computer-aided control system design tools
display/GUI design tools
source & AADL
computer system architecture and integration tools
source & AADL
hardware design tools
software reengineering tools
embedded computer system implementation

Figure 2: Domain-Specific and AADL Toolsets

figurable, safety-critical distributed systems that efficiently host both time-triggered and even t-triggered w orkloads. We will conclude this paper with a survey of some challenge problems, technical approaches, and preliminary results in these areas.

2 MetaH/AADL

The emerging SAE standard Avionics Architecture Description Language (AADL) is a language for specifying softw areand hardware arc hitectures for real-time, safet y-critical, scalable, embedded multiprocessor systems. The AADL allo ws dev elopers to specify how a system is composed from softw are components like processes and pac kagesand hardware components like processors and memories. Our MetaH/AADL toolset performs syntactic and semantic checks, compliance chec ks betw een specification and source code, schedulability analysis, reliabilit y analysis, partition isolation analysis, and generates/configures a middleware la y er that can be subjected to formal analysis using linear hybrid automata models. Figure 3 illustrates the current toolset.

Low-level soft w are constructs of the AADL describe source components written in a traditional programming language like C or Ada. The source components themselves come from domain-specific tools, or are hand-written, or are re-engineered from existing code. Subprogram and pac kage specifications describe important attributes of source modules such as the file containing the source code, nominal and maximum compute times on v arious kinds of processors, stack and heap requirements, mutual exclusion protocol to be used for shared pac kages,etc. Even t names and data buffer variables used to hold message values can appear within source modules and are described in the AADL specification. The current toolset will parse Ada source modules and check for compliance with their AADL interface descriptions.

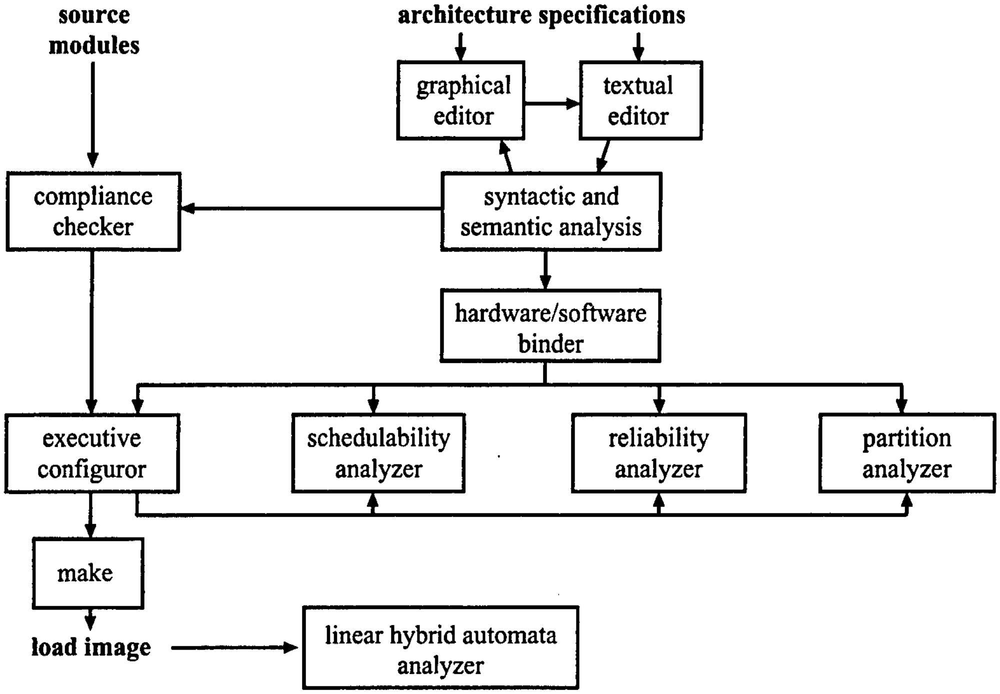


Figure 3: MetaH/AADL Toolset

The higher-level soft w are constructs of theAADL are processes, macros and modes. Processes group together source modules that are to be scheduled as either periodic (time-triggered) or aperiodic (even t-triggered) processes. A process is also the basic unit of securit yand fault con tainment, and memory protection and compute time enforcement may be provided if the target RTOS provides the needed support features. Macros and modes group processes, define connections betw een data and even t ports, and define bindings betw een objects that are to be accessable between processes. The difference is that macros run in parallel with each other, while modes are mutually exclusive. Event connections between modes are used to define hierarchical mode transition diagrams, where mode changes at run-time can stop or start processes or c hange connections.

The AADL also allo ws hardware arc hitectures to be specified using memory, processor, channel, and device components grouped into systems. Hardware objects may ha vedata and even tports and pac kages in their interfaces. Soft w are and hardware data and even t ports can be connected to softw aredata and event ports, and softw are components can access hardware pac kages(which provide hardware-specific APIs). Hardware descriptions iden tify(among other things) hardware-dependent source code modules for device driv ers, and code to provide a standard interface betw een automatically composed applications and the underlying RTOS.

Both graphical and textual specification is supported. The tw o can be mixed (part of a specification can be maintained textually and part graphically), and the toolset can translate graphical to textual and vice ersa. This is convenient in a soft w are and systems integration tool, since different parts of a specification may be produced by different groups or automatically generated b y different domain-specific tools.

A simple softw are/hardware binding tool assigns to hardware those softw are objects in a specification that are not explicitly assigned, possibly subject to user-specified constraints.

An executive configuration tool automatically produces the "glue" code needed to compose the various source modules to form the overall application. The resulting tailored middleware is responsible for process dispatching, event and message passing, mode changing, etc. There is a makeh tool that performs all the comples and links needed to produce a loadable image for eac h processor specified in the system.

The design schema for the configured executive is based on preemptive fixed priorit y scheduling theory. Using AADL specifications of process period, preperiod deadline, criticalit y, and precedence constrain ts, the executive generator derives priorit y, period transformation, and dispatch and time slice refill information used in data tables and dispatching code[11]. Data connection specificationand process timing information are used to schedule and generate code to move data betw een processes' data buffer variables. Message-passing code includes fault-handling constructs and is scheduled to meet hard real-time communication deadlines in multi-processor systems. Code to vector events to dispatch aperiodics or to trigger mode changes, and code to manage mode changes, is also generated.

Using information contained inthe AADL specification and produced b y the executive generator, the schedulability modeler generates a detailed preemptive fixed priority schedulability model of the application. The model includes middleware sc heduling and communication ov erheads as well as application workloads. In support of traceability, a h uman-readable form of themodel is written as w ell as the results of analyzing the model. The schedulability analysis algorithm we currently use is an extension of the exact characterization algorithm that that can perform certain kinds of parametric analysis[21].

The executive code generated from a MetaH specification may enforce integrated modular avionics partitioning (protected address spaces, process criticalities, enforced compute time limits, capability lists for runtime services). Source objects may be annotated with a safety level determined during system safety hazard analysis[1] (required application code v erification activities and hence the degree of assurance depend on the assigned safet y lev el). The tool chec ksto insure that correct operation of an object cannot be affected by any error in any other object having a lower safety lev el. F or example, an object with a high safety level should not depend on data from an object with a low safety level (unless the connection is explicitly annotated in the specification to allow this). The deadline of a process with high safety level must be guaranteed even if processes with low safet y lev els exceed their stated compute times.

The reliability and linear hybrid automata analysis tools will be discussed in later sections that describe recen t researc h activities.

3 Research

Our long-term goal is a language and computationally efficient toolset that support the development of embedded systems that are distributed, dynamically reconfigurable, fault-tolerant, support periodic (time-triggered) and aperiodic (even t-triggered) task models with complex inter-task interactions, make efficient use of resources, and are verifiable to the highest levels of system safety and design assurance. In the following sections we will cite some challenging problems in these areas and outline some of the approaches we are pursuing to deal with them.

3.1 Decomposition Scheduling

The distributed scheduling problem for systems that host periodic feed-back control applications is different than the multi-media scheduling problem. Tight end-to-end latencies comparable to task periods must be guaranteed. Often no loss of data will be tolerated. Solutions may need to be verified to the highest levels of assurance, which in practice means schedulability analysis must be av ailable. Our notional set of requirements is

- high achieveable hardware utilization, e.g. ov er 90% processor and ov er 75% bus utilizations
- small end-to-end latencies, e.g. one sampling delay (one period)
- high assurance that deadlines will be met, e.g. formal schedulability analysis
- tractability for large systems, e.g. generate schedules for thousands of tasks and messages on hundreds of processors in tens of seconds, incrementally c hange a schedule in fractions of a second
- compatibility with COTS bus/network hardware, adaptable to differences in scheduling requirements for individual resources, adaptable to differences in redundancy management techniques and interconnect topologies

We hav e been exploring an approach we call decomposition scheduling, illustrated in Figure 4. The basic idea is to decompose the overall system scheduling problem into a set of individual resource scheduling problems, solve the individual problems, then combine the results of parametric schedulability analysis for the individual resources to obtain a better decomposition. Each individual resource scheduling problem consists of the tasks or messages allocated to that resource, together with release times and deadlines that are selected b y the decomposition algorithm. Once each resource has been scheduled, the results of parametric schedulability analysis (such as av ailable laxity and slack for the various tasks and messages) are used to pick a new set of release times and deadlines. The new deadlines and release times make the individual

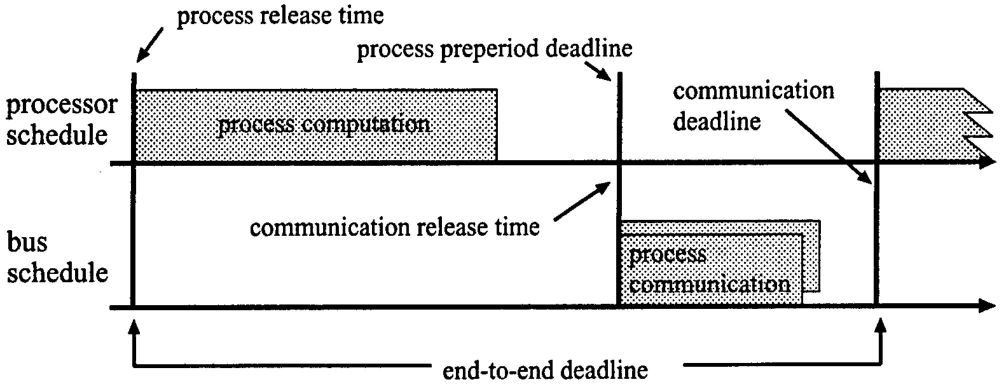


Figure 4: Decomposition Scheduling

scheduling problems easier for previously unschedulable resources at the expense of previously schedulable resources. The approach is iterative and continues until a solution is found or the solution does not change significantly between successive iterations.

The iterative nature of the approach makes it potentially adaptable to incremental rescheduling. Different resources can potentially be scheduled using different disciplines, as long as parametric schedulability analysis is availble for each discipline (our experiments used preemptive fixed priority).

We performed experiments using a highly abstracted workload from the B777 aircraft information management system (113 tasks, 6 processors, 50% bus utilization), and an early development workload from the Comanche mission equipment package (281 tasks, 24 processors). We constructed synthetic workloads by "connecting" multiple copies until the bus became unschedulable. Our prototype was able to schedule 6 connected Comanche systems (1686 tasks, 144 processors, 57% bus utilization) in 3 seconds of Sparc Ultra-2 CPU time. For comparison, the University of Maryland applied simulated annealing approach to a sanitized B777 problem and required 23 hours of CPU time[14]; the carefully tuned production scheduling tool required a few hours to produce a schedule for the fully detailed problem.

Our approach and preliminary results are similar to those of García and Harbour[15], although we use a different decomposition algorithm at each iteration. Our prototype also currently only schedules chains of length two (one task and its outgoing messages, with a direct bus available between sender and receiver).

3.2 Slack Stealing

Traditional control applications use periodic task and communication models, but many applications use event-triggered interactive task models. It remains a challenge to mix the two types of workloads in a way that guarantees periodic task deadlines, provides quick response times and high throughputs to the aperiodic tasks, and achieves high processor utilizations. Specific examples of such needs are the hosting of a message handling application, or a TCP/IP stack, or a Real-Time CORBA ORB, on the same system that also supports vehicle control applications.

We have been developing slack stealing methods to address this need. Slack stealing, as first proposed in [18], is a preemptive processor scheduling algorithm that delays the execution of high priority periodic tasks to improve the response times of aperiodic tasks while guaranteeing the periodic task deadlines. An on-line slack server determines at each event arrival (each request for slack CPU time) how big of a time slice can be immediately granted at a particular priority level without causing any periodic deadlines to be missed.

Table 1 contains data illustrating the difference between a background and (high priority) slack server when there is a single periodic task with (hyper)period $H = 10$ and compute time $C = 6$. The subscripts "bg" and "ss" refer to a background server and (high priority) slack server, respectively. The departure time of the n^{th} aperiodic task is denoted by d_n and the response time is $r_n = d_n - a_n$. Columns $r_{n,\mathrm{bg}}$ and $r_{n,\mathrm{ss}}$ shows the slack server providing smaller response times.

The background server processes aperiodic tasks only when there are no periodic tasks in the system. Suppose an aperiodic task α is in service, having completed x'_j of its execution when τ_n arrives at time $(n-1)H$. Task α will be preempted until time $(n-1)H+C$ while the periodic task executes, at which

id	a_n	x_n	$d_{n,\text{bg}}$	$r_{n,\text{bg}}$	$d_{n,\text{ss}}$	$r_{n,\text{ss}}$
1	1	1	7	6	2	1
2	3	2	9	6	5	2
3	6	2	17	11	11	5
4	8	1	18	10	12	4

T able 1:Fig 5 and 6 Sample Data ($H = 10, C = 6$)

time it will resume service with a remaining execution time requirement of $x_j - x'_j$. The background server timeline execution of the data in Table 1 is shown in Figure 5.

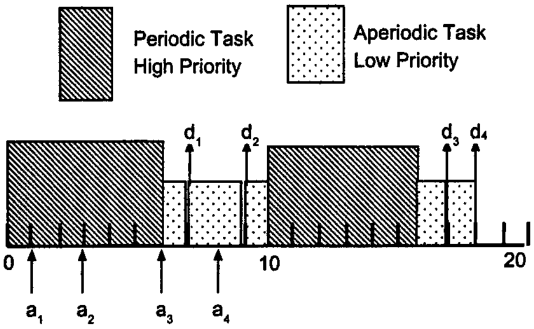


Figure 5: Background Server Timeline

The slack server processes aperiodic tasks at the highest priority as long as a periodic task will not miss its deadline. Let $C(t)$ be the amount of compute time an executing periodic task has consumed at time t, $0 \leq C(t) \leq \min(C, t)$. When $C(t) = C$, it remains at that value until $t = H$, then is reset to 0. $C - C(t)$ is the time required by the periodic task to complete by its deadline. $H - t$ is the time remaining in the current hyperperiod. The slack remaining in the current hyperperiod at time t is then $(H - t) - (C - C(t))$. In other words, an aperiodic task α arriving at time t to an empty aperiodic queue would complete without delay caused by the execution of a periodic task provided $x_j \leq (H-t)-(C-C(t))$. If $x_j > (H-t)-(C-C(t))$ then the aperiodic task would be blocked during the interval $[t, t + H - (C - C(t))]$ while the periodic task executes and completes exactly at its deadline. The slack server timeline execution of the data in Table 1 is shown in Figure 6. Note that the periodic task is blocking aperiodic tasks in the time interval $[7, 10]$ otherwise periodic execution occurs only when no aperiodic tasks are in the system.

When periodic tasks complete in less than their worst case execution time, the unused execution time can be reallocated at the priority at which it would have been executed. This form of slack is known as reclaimed slack (timeline slack is what is shown in Figure 6). Reclaimed slack is particularly important when safety-critical applications are present because extremely conservative worst-case compute times must normally be used to assure safety-critical deadlines. T able 2 is an augmented version of Table 1 for the execution timeline shown in Figure 7, where each execution of C actually completes after 4 units and 2 units are reclaimed.

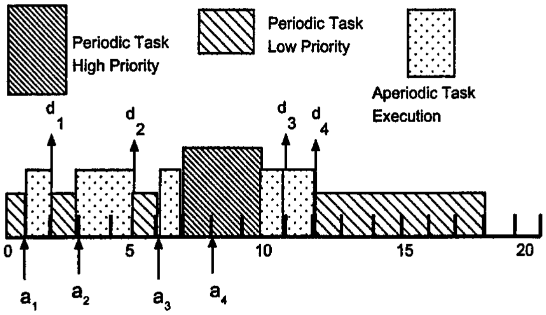


Figure 6: Slac k Serer Timeline

id	a_n	x_n	$d_{n,\text{ss}}$	$r_{n,\text{ss}}$
1	1	1	2	1
2	3	2	5	2
3	6	2	9	3
4	8	1	10	2
5	13	4	19	6
6	17	1	20	3

Table 2: Fig 7 Sample Data ($H = 10, C = 6, R = 2$)

Note that we differentiate between aperiodic execution on timeline versus reclaimed slack. Aperiodic tasks 3 and 4 now execute on reclaimed slack in contrast to Figure 6. Aperiodic task 5 begins its execution on timeline slack (in interval $[13, 16]$), and is then preempted by the periodic task to ensure its deadline. The periodic task completes early, allowing task 5 to finish its execution on reclaimed slack (in interval $[19, 20]$).

We have developed a variety of slack stealing techniques that are needed to use this technology in actual embedded systems. Our first real-time implementation was in MetaH[7]. We later adapted slack algorithms to support incremental processing[8] and then dynamic threads and time partitioning[9] in DEOS,

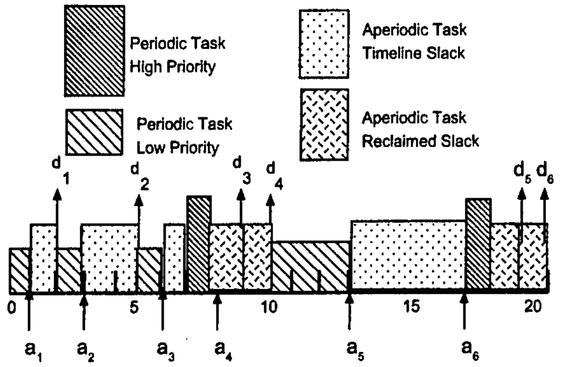

Periodic Task High Priority
Periodic Task Low Priority
Aperiodic Task Timeline Slack
Aperiodic Task Reclaimed Slack
d_1
d_2
d_3
d_4
d_5
d_6
0
5
10
15
20
a_1
a_2
a_3
a_4
a_5
a_6

Figure 7: Recalimed Slack Timeline

an RTOS contracted for use in six different FAA-certified commercial jets. We were able to obtain significant performance improvemerts relative to the deferred server algorithm that was originally used in DEOS. For example, the throughput of an FTP stack hosted on DEOS improved by a factor of 3 (when slak servers are used at one or both ends of a communication link with a handshaking protocol, throughput increases because less time is spent waiting for the parties to respond to each other). Perhaps more importantly, the bandwidth reserved for this application was reduced ly a factor of 7, dramatically increasing the available CPU utilization of the overall integrated system. Slack stealing has also been used to support incremental display tasks, where a minimum tolerable refresh update rate is guaranteed, with slack being used to almost always achieve a higher refresh rate. Our algorithms provide the safe co-hosting of Level E COTS FTP software with Level A safety critical software without compromising real-time performance measurements or achieveable CPU utilization.

We are currently investigating the applicability of, and extensions to, slack stealing for more complex models of task interaction, such as remote procedure calls and queueing networks; and application of some of these concepts to bus/network scheduling in distributed systems.

3.3 Response Time Analysis

In many application areas, such as telecommunications, performance is usually discussed in stochastic rather than deterministic terms. Averages alone are not sufficient, metrics based on knowledge of the response time distribution are desired (e.g. the expected percentage of requests that will be serviced within a stated deadline). A challenge problem is to analytically predict response time distributions for aperiodic tasks when they must share the CPU with periodic tasks. Our goals for analytic modeling of response time distributions in the presence of periodic tasks are

- efficient generation of aperiodic response time distribution approximations with confidence bands
- on-line parameter sensing/estimation for response time model validation, admission control, and dynamic reconfiguration
- analytic models that enable efficient bus/network scheduling for blending periodic feed-back control messages and event-triggered messages

We are investigating models for slack servers that execute at various priority levels. Figures 8 and 9 illustrate the predictions of some different analytic models plotted against simulation data for slack and background servers, respectively[10] ($H = 64$ ms, $C = 0.75H$, aperiodic traffic utilization is 0.2 with a mean service rate of 1 ms). We have developed new models called the long and intermediate hyperperiod models for slack and background servers (labeled LHM and IHM). For comparison purposes we also show an MM1 slack server model (labeled MM1), a heavy traffic background server model (labeled HTM), and the degraded server model (labeled DSM, which simply reduces the server speed by the fraction of the CPU taken by the periodic task). The simulation data points appear as a heavy line.

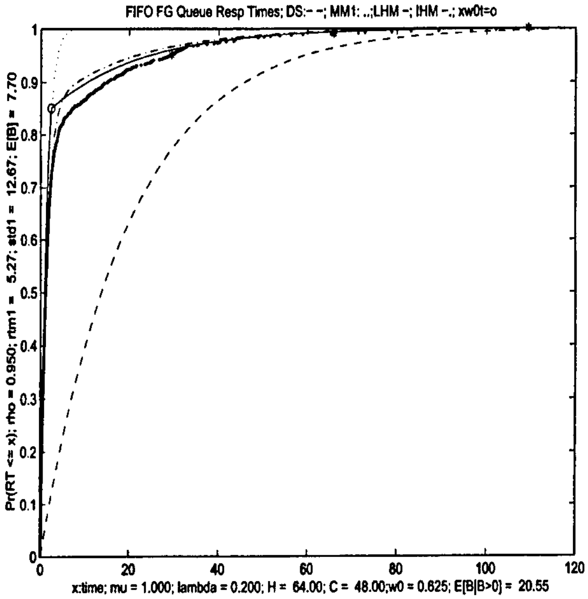


Figure 8: Slack Server Response Time

The observed aperiodic response time distribution for a slack server in Figure 8 lies completely above the

DSM response time prediction. In contrast, the observed aperiodic response time distribution when processed at background priority falls completely below the DSM response time distribution in Figure 9. For both server scheduling disciplines in the configurations shown, the long and intermediate hyperperiod models give estimates closer to the simulation data. Different models are better for different system configurations, and we have criteria for selecting the best model.

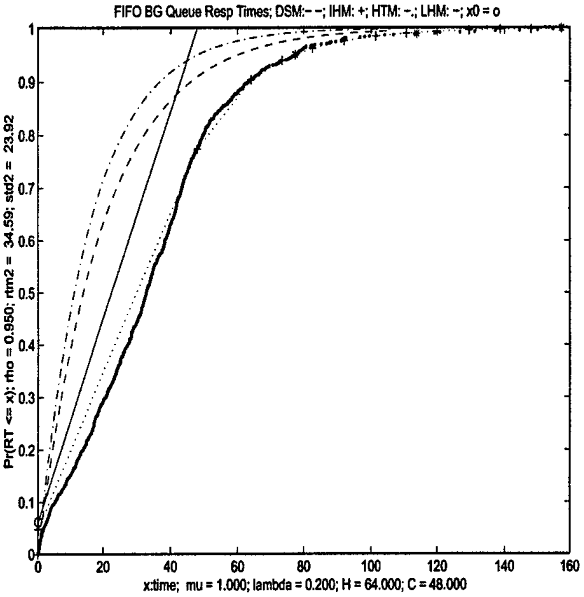


Figure 9: Background Server Response Time

We are also investigating the impact that periodic traffic patterns have on the delay distributions of the aperiodic traffic. In many commercial bus communication protocols with integrated periodic and event-triggered traffic, bus traffic is slotted and has designated start times for time-driven messages. Event-triggered traffic also has dedicated time slots, which are usually the remaining gaps not allocated to critical periodic messages. There may be no event-triggered messages waiting at the start of a preallocated event message time slot, or many messages might have been queued for a long time. In many regards, the study of these gaps on busses is analogous to the study of event-triggered task response times when run as background tasks on a CPU with predefined scheduling times for critical time-triggered periodic tasks. We have found that the spacing and size of the aperiodic gaps can have significant impact on the response delivery time distributions, suggesting it is possible to improve bus performance by appropriate scheduling of these gaps.

3.4 Hybrid Automata

Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors. One of our goals is to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory. For example, we would like to be able to model and analyze a system of tens of tasks on a few processors, where tasks may make remote procedure calls to each other, may have complex internal behaviors (multiple internal states with state transitions dependent on inter-task interactions), and have hard deadlines between specified pairs of state transitions.

At the implementation level, task schedulers and communication protocols are reactive components that respond to events like interrupts, message arrivals, service calls, task completions, error detections, etc. Another of our goals is to model and verify implementations of real-time functions. We would like to model important implementation details such as control logic and data variables. We would like the mapping between model and code to be clear and simple to better assure that the model really does describe the implementation. For example, we would like to be able to model and verify a real-time scheduler or a real-time bus driver.

We have been working with linear hybrid automata models of such systems[22]. Our experience suggests these are very powerful and natural models for very complex real-time system behaviors. However, computational intractability is currently a much more severe problem for hybrid automata model checking than it is for finite state model checking. We were able to overcome some of these problems by developing our own prototype reachability tool that uses new polynomial-time algorithms to compute hybrid state transitions, uses an oracle to concisely encode the scheduling semantics for a particular model, and does on-the-fly identification of reachable discrete states. For example, we were able to solve some problems having 100 times more discrete states than we could with other tools, although our prototype does not currently support rate ranges or provide parametric analysis. We also showed that the reachability problem becomes decideable under restrictions that are very reasonable for this problem domain.

We demonstrated these technologies by formally verifying key behaviors of the core scheduling and time partitioning modules of the MetaH executive. The standard executive library modules were modified by inserting calls to generate linear hybrid automata models of the code that manages basic scheduling and time partitioning functions (excluding slack stealing and dynamic reconfiguration features). Time-varying variables were used to model hardware timers and accumulated process compute time. Zero-rate variables were used to model some variables in the code. A complete linear hybrid automata model for this portion of the executive (about 1800 SLOC, about half of the executive) was automatically generated by executing each subprogram independently with test data, then subjected to a reachability analysis to verify some basic timing properties and assertions in the code. We analyzed a set of applications that was sufficient to achieve full coverage of the modeled code.

Our work to date suggests the technology has passed the threshold of utility for verifying implementations of certain real-time functions. However, significant improvements in analysis techniques are needed in order to analyze and verify the schedulability of complex real-time workloads of non-trivial size.

3.5 System Safety

Our MetaH toolset supports a construct called an error model, which allows users to specify sets of fault events and error states. An error model includes specifications of transition functions to define how the error states of objects change due to fault, error propagation and recovery events. An individual object within a specification can then be annotated to specify the error transition function and fault arrival rates for that object.

We have a prototype reliability modeling tool that generates a stochastic concurrent process reliability model[17, 20]. Error propagations between objects are modeled as synchronizations or rendezvous between stochastic concurrent processes. Each such propagation synchronization in the model can be controlled using an associated consensus expression, which can conditionally mask propagations depending on the current error states of selected objects. In the specification, user-supplied consensus expressions describe the error detection protocols that are implemented by the underlying source modules for a particular application. The reliability modeler uses the error model specifications and annotations to generate the object error state machines, and uses the consensus expressions and design structure to generate the propagation synchronizations between these object error state machines. A subset of the reachable state space of this stochastic concurrent process is a Markov chain that can be analyzed using existing tools and techniques[19]. We selected a stochastic concurrent process model because it allowed us to generate a hierarchical reliability model whose structure can be easily traced back to the original specification and vice versa.

We believe this work substantiates the basic concept of generating a reliability model from a design specification, but we have identified a number of shortcomings that must be addressed for production use[12]. Markov model generation and analysis is subject to state space explosion; features in the language to control abstraction in the generated model, and state space optimization methods in the analysis tool, are needed. Analysis results that are easily traceable back to the specification and include parametric sensitivity data are needed.

Markov reliability analysis and partition isolation analysis are only two types of analysis used in a comprehensive system safety program[23]. The language already contains some features for fault tree specification, but features are needed to capture the results of hazard analysis and failure modes and effects analysis and summary. All these different analyses are synergistic and related, and the analysis toolset should perform certain consistency checks between the different models and results. For example, basic events in a fault tree are assumed to be statistically independent, and all common cause analyses (such as partition isolation analysis) should check for the absence of common causes for pairs of basic events.

We close by noting that software safety standards require system safety program activities to be closely integrated with development activities, and require that safety data and analyses be clearly traceable to development work products such as designs and code[3, 2]. Design assurance standards require review and analysis in addition to testing, and encourage as high a degree of formal analysis as is practical[1]. The well-integrated and formalized development process and environment that we are pursuing can significantly contribute to meeting these requirements.

References

[1] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc., Washington D.C., December 1992.

[2] *Software System Safety Handbook*, Joint Software System Safety Committee, December 1999, www.nswc.navy.mil/safety/handbook.pdf

[3] *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Sys-*

tems and Equipment, SAE/ARP 4761, December 1996.

[4] *MetaH User's Guide*, Honeywell Laboratories, 3660 T echnologyDrive, Minneapolis, MN, www.htc.honeywell.com/metah.

[5] *Domain Modeling Environment*, Honeywell Laboratories, 3660 T echnologyDrive, Minneapolis, MN, www.htc.honeywell.com/dome.

[6] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE T ransactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[7] Pam Binns, "Scheduling Slack in MetaH," *R eal-Time Systems Symposium*, w ork-in-progress session, December 1996.

[8] Pam Binns, "Incremental Rate Monotonic Scheduling for Improv ed Control System P erformance," *R eal-TimeApplications Symposium*, 1997.

[9] P am Binns, "A Robust High-Performance Time P artitioning Algorithm;The Approach Taken in DEOS," to appear in the *20th Digital Avionics Systems Conferenc e,* Nov ember 2001

[10] Pam Binns, *Aperiodic R esponse Time Distributions in Queues with Deadline Guarante es for Periodic Tasks*, Ph.D. Thesis, Department of Statistics, University of Minnesota, October 2000.

[11] Pam Binns and Stev eV estal, "Message P assing in MetaH using Precedence-Constrained Multi-Criticality Preemptive Fixed Priorit y Scheduling," *Life Cycle Software Engineering Conferenc e,* Redstone Arsenal, AL, August 2000.

[12] Pam Binns, Stev eV estal, William Sanders, Jay Doyle and Dan Deavours, "MetaH/Möbius Integration Report," prepared by Honeywell Laboratories and University of Illinois Coordinated Science Laboratory, prepared for U.S. Army AMCOM Soft w areEngineering Directorate, April 2000.

[13] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitativ e Characteristics of Finite-State Real-Time Systems," *R eal-Time Systems Symposium*, December 1994.

[14] Shent-Tzong Cheng and Ashok K. Agrawala, "Allocation and Scheduling of Real-Time Periodic T asks with Relative Timing Constraints," Univ ersity of Maryland Department of Computer Science Technical Report, 1993.

[15] José Javier Gutiérrez García and Michael González Harbour, "Optimized Priorit yAssignment for T asks and Messages in Distributed Hard Real-Time Systems," *Third Workshop on Parallel and Distributed Real-Time Systems*, April 1995.

[16] Bruce Lewis, "Soft w areP ortabilit yGains Realized with MetaH, an Avionics Architecture Description Language," 18th *Digital A vionics Systems Conference*, St. Louis, MO, October 24-29, 1999.

[17] Holger Hermanns, Ulrich Herzog and V assilis Mertsiotakis, "Stochastic Process Algebras as a T ool for Performance and Dependability Modeling," *Pr oceedings of the IEEE International Computer Performance and Dependability Symposium* (IPDS'95), April 24-26, 1995, Erlangen, Germany.

[18] J. P .Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Aperiodic Tasks in Fixed-Priority Preemptive Systems," *IEEE Real-Time Systems Symposium*, December 1992.

[19] W. H. Sanders, W. D. Obal, M. A. Quershi and F. K. Widjanarko, "The UltraSAN Modeling Environment," *Performance Evaluation Journal*, vol. 25 no. 1, 1995.

[20] F rederic kT. Sheldon, Krishna M. Kavi and F arhad A. Kamangar, "Reliability Analysis of CSP Specifications: A New Method Using P etri Nets," *Proceedingsof AIAA Computing In A erosp ace*, San Antonio, TX, March 28-30, 1995.

[21] Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, April 1994.

[22] Steve V estal, "Modeling and V erificationof Real-Time Soft w areUsing Extended Linear Hybrid Automata," *NASA L angley F ormal Methods Workshop*, June 2000, shemesh.larc.nasa.gov/fm/Lfm2000/Proc/

[23] Steve Vestal, "MetaH Avionics Architecture Description Language Soft w areand System Safet y and Certification Study," prepared by Honeywell Laboratories, prepared for U.S. Army AMCOM Softw are Engineering Directorate, March 2001.

Design Models for Components in Distributed Object Software[1]

X. Xie and S. M. Shatz

University of Illinois at Chicago

Abstract

Component-based software development has many potential advantages, including shorter time to market and lower prices, making it an attractive approach to both customers and producers. However, component-based development is a new technology with many open issues to be resolved. One particular issue is the specification of components as reusable entities, especially for distributed object applications. Specification of such components by formal methods can pave the way for a more systematic approach for component-based software engineering. This paper discusses an approach for blending Petri net concepts and object-oriented features to develop a specification approach for distributed component software systems. In particular, a scheme for modeling behavior restriction in the design of object systems is presented. A key result of this work is the definition of a "plug-in" structure that can be used to create "subclass" object models, which correspond to customized components.

Keywords: Distributed Software, Modeling, Object Design, Petri Nets

1. INTRODUCTION AND MOTIVATION

There is significant interest in using components in software development. Specification and implementation of a system in terms of existing and/or derived components can dramatically decrease the time required for system development, increase the usability of resulting products, and lower production costs [8]. However, component-based development is still immature, with a lack of established procedures and support from formal modeling.

Reuse principles have typically placed high demands on reusable components. Such components need to sufficiently general to cover the different aspects of their use, while also being simple enough to serve a particular requirement in an efficient way. This has resulted in a situation where developing a reusable component may require significant effort. Reuse can be aided by customization that applies constraints in situations where the functionality of a "base component" is more general than is actually needed, or when some base-component features exhibit characteristics not suitable for a particular application. Thus, the component's behavior must be restricted before it can be reused in a new design.

One potentially efficient and natural technique to support constraints is restriction inheritance [2]. Restriction inheritance defines a subclass that constrains the behavior of a superclass. This is in contrast to augment inheritance, where a subclass augments, or extends, a superclass. Since subclassing by restriction often conflicts with the semantics and intention of inheritance, where an instance of a subclass should be an instance of the superclass and should behave like one, some researchers have suggested that restriction inheritance be avoided [8]. But, in our own experience, which does involve

[1] This material is based upon work supported by, or in part by, the U.S. Army Research Office under grant number DAAD19-99-1-0350 and by NSF under grant number CCR-9988168.

development of commercial component-based software, we have observed benefits of restriction inheritance for customizing components.

To develop a systematic design process with the capability for automated simulation and analysis, it is valuable to define a design method's syntax and semantics in terms of some formal notation and method. For engineering of distributed object systems, it is desirable for the formalization to provide a simple and direct way to describe component relationships and capture essential properties like nondeterminism, synchronization and concurrency. Petri nets [5] are one formal modeling notation that is in many ways well matched for general concurrent systems. In particular, the standard graphical interpretation of Petri net models is appealing as a basis for a design notation. In this paper we introduce a model called a State-Based Object Petri Net (SBOPN), which is developed from the basic idea introduced in [6]. Here we extend the basic SBOPN model to directly support restriction inheritance modeling for the purposes discussed earlier. SBOPN is most similar in spirit to Lakos' Language for Object Oriented Petri Nets, LOOPN [4]. LOOPN's semantics are richer, but SBOPN provides a more specific, and thus more intuitive, notation for capturing the behavior of distributed state-based objects. Like LOOPN, SBOPN is based on a generalized form of Petri net called colored Petri nets [3]. Another language, CO-OPN/2 [1], uses high-level Petri nets that include data structures expressed as algebraic abstract data types and a synchronization mechanism for building abstraction hierarchies to describe the concurrency aspects of a system. CO-OPN/2 is a general model that focuses on concurrency. SBOPN focuses more on the architectural modeling of state-based systems; thus it is simpler and domain-specific.

2. AN EXAMPLE AND INTRODUCTION TO SBOPN MODELING

Consider the classic example of a system that uses a bounded buffer to temporarily hold items, such as messages. In our example, there exists an operator to enable and disable the buffer, in addition to the standard producer and consumer components. The four system components – buffer, producer, consumer and operator – operate asynchronously and only interact via messages initiated by the producer (*put* message), consumer (*get* message) or operator (*enable* and *disable* message). At any point in time, the buffer should be in one of four states: *Empty*, *Full*, *Partial* (means *Partially Full*) or *Disabled*. Depending on its state, the buffer may or may not be able to accept the messages *put*, *get*, *disable* and *enable*. When the buffer is in *Empty* or *Partial* state, it can accept the *put* message and change to *Partial* or *Full* state. When it is in *Partial* or *Full* state, it can accept the *get* message and change to *Empty* or *Partial* state. When it is in any state except the *Disabled* state, it can accept the *disable* message and change to the *Disabled* state. Finally, when it is in the *Disabled* state, it can accept the *enable* message and change to its previous state (before it was disabled): *Empty*, *Partial* or *Full*. To simplify the example, we simply assume that after accepting a *disable* message, the buffer is reset to *Empty* state.

To model state-based systems, such as this buffer system, we use State-Based Object Petri Nets (SBOPN) [6]. This can be viewed as a special purpose form of (Colored) Petri net. Lack of space prevents us from giving an overview of Petri nets here; we refer the reader to a reference like [5] for such information. Figure 1 shows a simple SBOPN model of the system we have described above. Notice that there are separate models for the buffer, producer, consumer and operator objects. These components are called State-Based Petri Net Objects (SBPNO) and the methods of objects are represented by *shared transitions*. For example, the *put* method is represented by a shared transition used by the buffer object and the producer object. The *system model* is called a State-Based Object Petri Net (SBOPN). To informally highlight some key features of the SBOPN model, let us consider the buffer object. There is an arc from the place p_1 to the shared transition *put*.

The token labeled D in p_1 is called a *state token*, and D is the current *state-value* of this state token. This represents that the current state of the buffer is *Disabled*. The label *{Empty, Partial}* for the arc *(p_1, put)* shows that the *put* transition has the potential to fire only when the buffer is in the *Empty* or *Partial* state. This arc label is called a *state filter*. When all the input places of a transition satisfy the corresponding state filter, that transition is enabled. The arc from the transition *put* to the place p_1 is also labeled. This arc label *(p_1, F1)* is called a *state-transfer tuple*, where p_1 is called a *state-transfer place* and *F1* is called a *state-transfer function*. This tuple determines the possible state(s) the buffer can be in after the *put* method is processed. The input value of a state-transfer function is the state-value of the state token consumed from the associated state-transfer place. In this simple example, the buffer can have the following changes due to the *put* method: from *Empty* to *Partial*, from *Partial* to *Partial*, or from *Partial* to *Full*. The state-transfer function *F4* indicates that a call to the *disable* method results in the buffer transitioning to the *Disabled* state, regardless of the state-value of the token consumed from place p_1.

Now, consider a need to customize this general buffer component for use in a more restricted application. First, assume the new buffer component should not allow the disable operation. Second, to ensure tighter synchronization on producer and consumer components, the new buffer component should behave as a simple capacity-1 buffer. Thus, only when the buffer is in the *Empty* state, instead of both *Empty* and *Partial* states, should it accept a *put* message. We call this new buffer a "disable-free synchronous buffer." To model a new system that uses a disable-free synchronous buffer, we could just redesign the system in Figure 1 to create a new model. But, there are disadvantages that can result from a "re-design." First, creating the new buffer might change the interface of another class, the operator. This conflicts with the basic modularity principle of object-design. This is an important issue, especially when it comes to consideration of model synthesis and reuse. Second, the redesign might result in a change in the state filter for arc *(p_1, put)* in the general buffer class -- from *{Empty, Partial}* to *{Empty}*. But, such a change makes it now difficult to directly identify that the new object is actually one of many possible behaviorially restricted objects derived from a common object – borrowing from object terminology, we can think of these restricted object components as representing subclass objects of a superclass object. We will revisit this issue in Section 3.

We propose to model restriction inheritance by the simple addition of a "plug-in" structure to a superclass model. In other words, we want to limit the behavior of the superclass object by adding some control structure to the superclass model. Actually, this is very natural from the view of control theory since control systems limit the behavior of a system by adding some controller logic. For example, [9] describes a method for constructing a Petri net controller for a discrete event system modeled by a Petri net.

3. MODELING RESTRICTION SUBCLASS OBJECTS

In Section 2 we informally introduced the SBOPN modeling notation via an example. Now we can formally define this notation and discuss how to derive design models for subclass objects.

Definition 1 (SBPNO): A *State-Based Petri Net Object* is a 7-tuple, SBPNO = *(Type, NG, States, sp, ST, SFM, STM)*, where

- *Type* is an identifier for the object's type (or class).
- *NG = (P, T, A)* is a net graph, with P as a finite set of nodes, called Places; T as a finite set of nodes, called Transitions, disjoint from P, i.e., $P \cap T = \phi$; and $A \subseteq (P \times T) \cup (T \times P)$ as a set of arcs, known as the flow relation.

- *States* is a finite set of distinct states that define the possible states of the SBPNO. A token (as in standard, or colored Petri nets) may have associated with it a state-value, which is one of the elements of *States*.
- $sp \in P$ is called a state place. The value associated with the token in this place indicates the current state of the SBPNO.
- $ST \subseteq T$ is a set of shared transitions. A shared transition in a SBPNO is a transition that is shared with other SBPNOs. Shared transitions model the acceptance of a message from other SBPNOs or the sending of a message to other SBPNOs.
- $SFM: (A \cap (P \times T)) \rightarrow 2^{States}$ is a state-filter mapping, where 2^{States} is the power set of *States*. This mapping maps each place-to-transition arc to a state filter. The basic purpose of the state filter mapping is to ensure that only those tokens that have a state-value representing one of the states in the state filter can pass (i.e., be consumed by a transition) via the corresponding arc.
- $STM: (A \cap (T \times P)) \rightarrow P \times STF$ is a state-transfer mapping, where STF is the set of state-transfer functions, $STF = \{stf \mid stf: States \rightarrow 2^{States}\}$. This mapping maps each transition-to-place arc (t, p) to a state-transfer tuple (p', stf), where $p' \in \{p \mid (p, t) \in A\}$ is called the state-transfer place and stf is called the state-transfer function. The basic purpose of the state-transfer mapping is to allow the firing of transition t to map the state-value of the token consumed from place p' into a set of states, which represents the possible state-values that can be associated with the token deposited into the output place via the corresponding arc.

To simplify SBPNO models, implicit (default) state filters and implicit state-transfer tuples are allowed. For an implicit state filter, the state-filter is *States*. Note that in Figure 1, the state filters are implicit in the producer, consumer, and operator objects. An implicit state-transfer tuple can be used only when the output place associated with the arc is an input place of the transition associated with the arc – the arc is part of a self-loop. The state-transfer place is the place in the self-loop. We also require an implicit state-transfer function's output to be the state-value of the token removed from the place in the self-loop. Due to the simplicity of the producer, consumer, and operator object models, the state-transfer tuples are also implicit.

Now we can identify properties of a restriction subclass and present the definition of a restriction subclass model, i.e., a model for a restriction subclass object. First, the methods of a restriction subclass object should be a subset of the methods of the superclass object. Second, the externally observable behavior of a restriction subclass object should be observable in the behavior of the superclass object. Thus, any firing sequence of a SBPNO subclass model should be a firing sequence of the superclass model when we only consider the shared transitions. A particular restriction subclass model must be defined in terms of some particular superclass model and some specific method restrictions. These restrictions are captured by a restriction function, as defined next.

Definition 2 (Restriction Function): Let $N_1 = (Type_1, NG_1, States_1, IS_1, Stoken_1, ST_1, SFM_1, STM_1)$ be a SBOPN, and let function $f: SF_1 \rightarrow 2^{States1}$, where SF_1 is the domain of SFM_1, and $2^{States1}$ is the power set of $States_1$. The function f is called a *restriction function* for N_1 if and only if f satisfies: $\forall sf_1 \in SF_1, f(sf_1) \subseteq sf_1$.

Applying f to the state filters of N_1 creates a new model, which we denote as N_1/f. It can be shown that N_1/f is a restriction subclass model of N_1, but note that N_1/f features the two disadvantages discussed earlier in Section 2. Our goal is to create a "plug-in" structure that can be added to a superclass model causing it to have the same behavior as N_1/f but avoiding these disadvantages. Such a plug-in structure must be able to control the firing of some shared transition t. This is accomplished by using a so-called "control place" as the heart of the plug-in structure. The control place must ensure that the state-value of a token in the control place "tracks" the state-value of a token in one of the input places p to the transition t. We call such a place p the "controlled place."

Definition 3 (Control Place): Let $N = (Type, NG, States, ST, SFM, STM)$ be a SBPNO, and p_1 and p_2 be two places of N. We say that p_2 is a control place for p_1 (p_1 is a controlled place) if and only if:

1) $(ST \cap p_2^\bullet \neq \emptyset) \wedge (ST \cap p_2^\bullet \subseteq ST \cap p_1^\bullet)$ (Note: $p_1^\bullet$ is the set of output transitions of the place p_1).

2) For any shared transition $t \in (ST \cap p_2^\bullet)$, the associated state filter for the arc (p_2, t) is a subset of the state filter for the corresponding arc (p_1, t).

3) For any reachable marking M' from M, which satisfies $M(p_1) = M(p2)$, and any transition $t \in (ST \cap p_2^\bullet)$, if t fires under M', then the tokens consumed by t from p_1 and p_2 should have the same state values.

3.1 Basic Plug-in Design

Since our goal is to ensure that the state-marking of a control place "tracks" that of the controlled place, we can copy the token of a controlled place into the control place, but we must be sure that this copying occurs before allowing these places to enable any shared transition. We call this type of control place a "refreshing place" since it gets refreshed (i.e., the state-value of its current state token is updated) each time the state-value of the token in the corresponding controlled place changes. Figures 2, 3 and 4 illustrate this idea by a simple example. In Figure 2, we present a SBPNO model for an object $C1$. Now, suppose that we want to derive a model for a restriction subclass object $C2$ that has the property that t_1 can be enabled only when the object is in the state a – instead of either state a or b, as in the superclass object $C1$. We need t_2 to remain enabled in the a-state.

To model this subclass object, we create a new place p_2 (see Figure 3) as a control place. Transition t_3 is introduced for the purpose of copying the state token from p_1 to p_2. The state filter associated with p_2's connection to t_1 is $\{a\}$. However, under the general firing rule that controls the behavior of a SBPNO, we cannot guarantee that the tokens in p_1 and p_2 are of the same value when t_1 is enabled. For example, in Figure 2, suppose p_1 has initial state a, then the firing sequence is $t_1^\bullet t_2 t_1^\bullet$. Now consider Figure 3, where both p_1 and p_2 have initial state a. Once t_2 fires, p_1 has state b, while p_2 still has state a. If t_3 does not yet fire, p_1 and p_2 have different states, but t_1 is still enabled. As a result, we could get the same firing sequence as $C1$, $t_1^\bullet t_2 t_1^\bullet$. However, $C2$ is supposed to only allow the restricted firing sequence $t_1^\bullet t_2$, where we ignore the internal transition t_3 in the firing sequence. So the construction in Figure 3 does not yet provide for a proper modeling of the control place.

The problem is that when t_2 fires, the token in p_2 remains unchanged and thus is not "tracking" the marking of p_1. To solve this problem, we need to force t_3 to fire immediately after t_2 fires, i.e., to refresh p_2 immediately. This is accomplished by using a special form of Petri net arc called an activator arc [7]. An activator arc can be used to connect a place to a transition. For nets with activator arcs, the transition firing rules are as follows: 1) Those enabled transitions with activator arcs have the highest priority, and 2) A transition that has activator arc input(s) cannot fire twice in succession for the same input marking, i.e., the net's marking must be modified in some manner before the transition can fire again. For example, in Figure 4, t_1, t_2 and t_3 are enabled, but t_3 has an activator arc (denoted by the arc with a solid bubble), so it fires first. After firing t_3, we get the same marking, so t_3 cannot fire again. As a result, only t_1 or t_2 can next fire. Now, if t_1 fires, because the marking remains unchanged, we have the same situation as before t_1 fires. But if t_2 fires, both t_1 and t_3 are enabled. Since the marking has changed, only t_3 can fire, which copies the token *b* from p_1 to p_2, i.e., p_2 is refreshed. This copying of the state-value from *p1* to *p2* is due to the state-transfer function *F3*. Note that t_1 is not enabled any more after t_3 fires. As we can see, p_2 now serves as a proper control place to ensure we have only one firing sequence $t_1^*t_2$ (again, ignoring the internal transition t_3 in the firing sequence).

```
Algorithm 1: Model a restriction subclass object by use of plug-in structures
Input:  A SBPNO $N_1 = (Type_1, NG_1, States_1, ST_1, SFM_1, STM_1)$.
        A restriction function (see Definition 2), $f: SF_1 \rightarrow 2^{States1}$
Output: A restriction subclass model $N_2$ of $N_1$ ($N_2$ has the same externally observable behavior as the model $N_1/f$
identifiedearlier).
1)      Make a copy $N_1$. Call this new model $N_2$ and let $N_2$ be the source net for the following step:
2)      For each transition $t$ in $ST_1$:
        For each $p_1 \in t^{\bullet}$, let $S1$ be the state filter for the arc $(p_1, t)$. If $S2 = f(S1)$ is a proper subset of $S1$, i.e., $S2 \neq S1$, then
        create a control place $p_2$ of $p_1$ by applying the following steps:
        A.    Create a refreshing place $p_2$ of $p_1$. (* An algorithm for creation of a refreshing place is given in [10]. The
        basic idea can be understood by observing that the application of this step to the SBPNO in Figure 2 creates part of
        the SBPNO shown in Figure 4 – all of the model except the arcs $(p2, t1)$ and $(t1, p2)$, and the state-filter $\{a\}$ for the
        arc $(p2, t1)$. *)
        B.    Add an arc $r_1$ from $p_2$ to $t$. Use $S2$ as the state filter for $r_1$. Add an arc $r_2$ from $t$ to $p_2$
        End For
        End For
```

The initial marking of a subclass model created by Algorithm 1 is determined by the initial marking of the source superclass model. All places except the created control places have the same initial marking as in the superclass model. The control places take on the same initial marking as their corresponding controlled places. As an example, applying Algorithm 1 to the SBPNO in Figure 2 creates the SBPNO shown in Figure 4. In this case, N_1 is the model shown in Figure 2 and the restriction function *f* is defined as $f(\{a, b\}) = \{a\}, f(\{a\}) = \{a\}$. Note that the structure within the dashed box in Figure 4 is the plug-in structure.

3.2 Switchable Plug-in Structures

One advantage of Algorithm 1 is that the plug-in structures created are potentially controllable. By controllability we mean that a switch can be added to the structure to control its activity, i.e., the switch can be used to "turn on" or "turn off" the functionality of the plug-in structure. We call such a plug-in a "switchable plug-in." A switchable plug-in offers a key advantage: It allows an SBOPN model to represent a family of restriction subclass models, corresponding to a family of components.

To transform a plug-in structure into a switchable plug-in, a new place node must be added. For example, Figure 5 shows the same model as Figure 4, but with a switchable plug-in. Place p_3 serves as this new switch place. When there is a token in the switch place p_3, the "plug-in" structure is active. In this case, the plug-in behaves as before we introduced the switch place, i.e., like Figure 4. But when there is no token in p_3, the transition t_3 will never be enabled. So, in this case, the model behaves as before we introduced the plug-in, i.e., like Figure 2. Notice that we have introduced a new state value called *internal* to the state set. Although it is possible to create the switching capability for this particular example without introducing this new *internal* state, use of this special state is required for creating general-purpose switchable plug-ins. In general, to model a restriction subclass using switchable plug-ins, we can use Algorithm 1 with the following two simple modifications: 1) For each plug-in, create a switch place (connected to/from the transition for the refreshing place); 2) For each plug-in, modify the state filter (for the arc from the control place to the restricted transition) to include the state *internal.*

As an example, let us revisit the buffer example from Section 2. Now, the modified algorithm mentioned above can be applied to the model in Figure 1 to create a model for a "disable-free synchronous" buffer. The resulting model would employ two switchable plug-ins – connected to the method transitions *disable* and *put*. The initial marking of all places belonging to the plug-ins is *internal.* For the plug-in associated with the disable method, the state filter is set to {internal}. Thus if the plug-in is "turned-on" (by marking its switch place), the disable method will become inactive. For the plug-in associated with the put method, the state filter is set to {i, Empty}, so that the put method is only active when the buffer is in the empty state. Due to a lack of space, we cannot show this buffer model, but it is given in [10]. Of particular interest is the observation that this one subclass model actually represents a family of buffer types. The binding of the model to a specific buffer behavior is accomplished by varying the initial markings of the switch places (which we can all the *Disable-switch* and the *Put-switch*). The following table illustrates the options:

Disable-switch	*Put-switch*	*Model*
Marked	Marked	A "disable-free synchronous" buffer
Marked	Unmarked	A "disable-free" buffer
Unmarked	Marked	A "synchronous" buffer
Unmarked	Unmarked	A general buffer (as in Fig. 1)

The ability to model a family of components can be very helpful for component-based development. It supports flexible analysis of varying configurations of customized components in the design phase, which can reduce the overall cost of development.

As a final comment, we note that we did not explicitly discuss any issues regarding model analysis. But, we can note that the SBOPN model, including plug-in structures, can be transformed to standard Colored Petri nets, and then to ordinary Petri nets. Thus, the discussed modeling approach can leverage on existing and developing techniques for analysis, including work aimed at the state-space explosion problem. Further details are provided in [10].

4. CONCLUSION AND FUTURE WORK

One challenge in component-based software engineering is to find techniques and tools that are effective in aiding the specification and design of component-based systems. One way to increase the effectiveness of these design techniques is to employ formal methods that provide a well-defined design notation and support design analysis. In this paper, we have discussed our research to blend Petri net concepts and object-oriented design in order to develop a design approach for component-based software systems development. A unique feature of this work is the idea of a "plug-in" control structure to allow for modeling restricted behavior on the part of object models. We discussed how this can support the modeling of customized components. One area for future work is to develop some prototype tools that can be used to automate the creation of SBOPN designs for complex systems, including support for synthesis and management of customizing general components to particular components. In addition, we plan to investigate capabilities for other forms of inheritance modeling.

REFERENCES

[1] Biberstein, O., Buchs, D. and Guelfi, N. "CO-OPN/2: A Concurrent Object-Oriented Formalism," *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Canterbury, UK, July 21-23 1997*, Chapman and Hall, London, 1997, pp. 57-72.

[2] Booch, G. "Object-Oriented Analysis and Design, with Applications (2 nd ed.)," *Benjamin/Cummings*, San Mateo, California, 1994.

[3] Jensen, K. "Coloured Petri Nets: A High Level Language for System Design and Analysis," *Advances in Petri Nets 1990*, G. Rozenberg (Editor), in *Lecture Notes in Computer Science*, 483, Springer-Verlag, 1990.

[4] Lakos, C.A. "Pragmatic Inheritance Issues for Object Petri Nets," *Proceedings of TOOLS Pacific '95 Conference* (The 18th Technology of Object-Oriented Languages and Systems Conference), C. Mingins, R. Duke, and B. Meyer (Eds), Prentice-Hall, 1995, pp. 309-322.

[5] Murata, T. "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, April 1989, pp. 541-580.

[6] Newman, A., Shatz, S.M. and Xie, X. "An Approach to Object System Modeling by State-Based Object Petri Nets," *Journal of Circuits, Systems, and Computers*, Vol. 8, No. 1, Feb. 1998, pp. 1-20.

[7] Ramaswamy, S. and Valavanis, K.P. "Hierarchical Time-Extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Hierarchical Systems," *IEEE Transactions on Systems, Man and Cybernetics*, Feb. 1996.

[8] Szyperski, W. "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 1998.

[9] Yamalidou, K., Moody, J., Lemmon, M. and Antsaklis, P. "Feedback Control of Petri Nets Based on Place Invariants," *Automatica*, Vol. 32, No. 1, pp. 15-28, 1996.

[10] Xie, X. and Shatz, S.M. "Design Support for State-Based Distributed Software," Technical Report, Concurrent Software Systems Laboratory, Dept. of EECS, UIC, 2000.

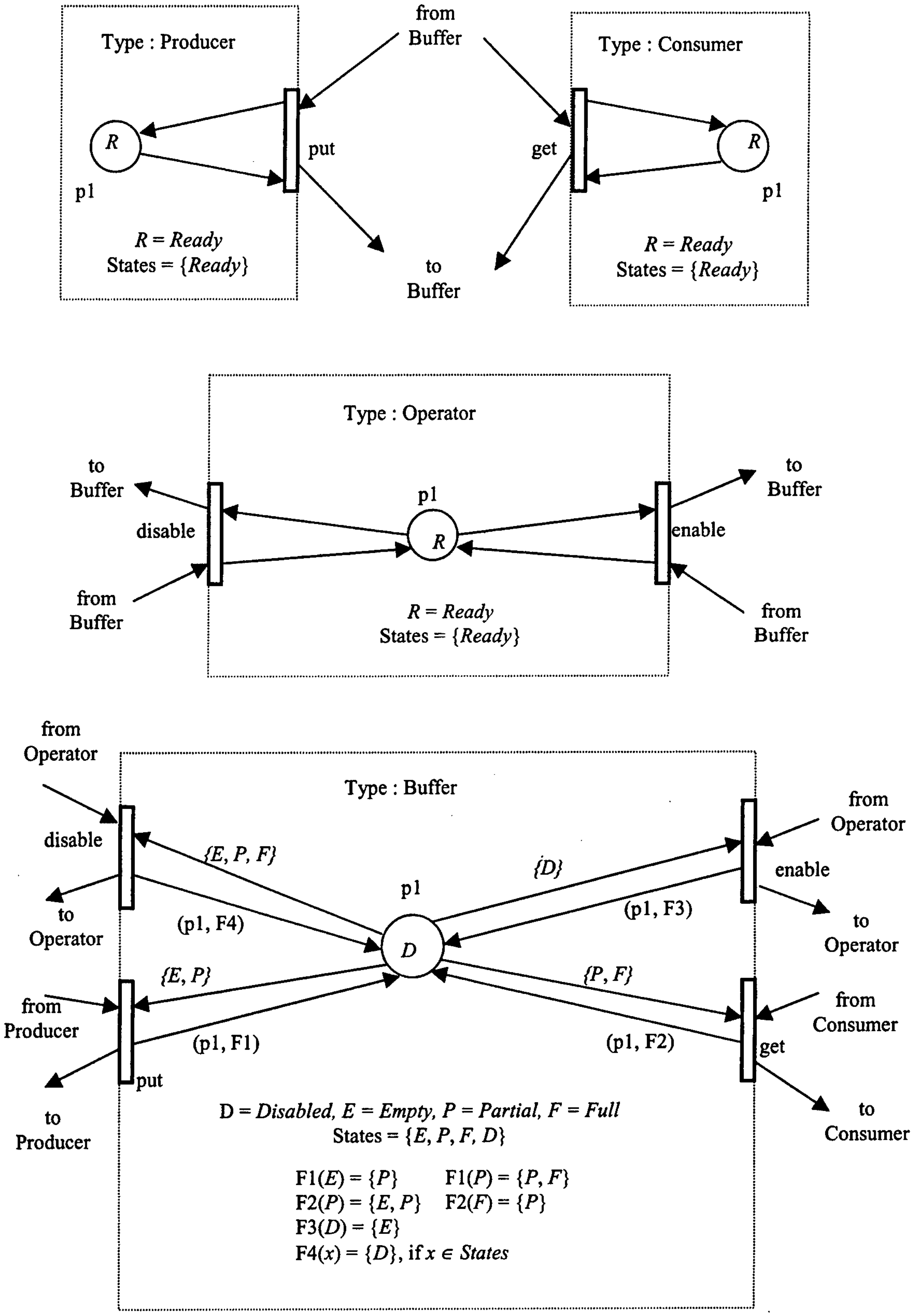


Figure 1. A SBOPN for the buffer, producer, consumer, and operator system

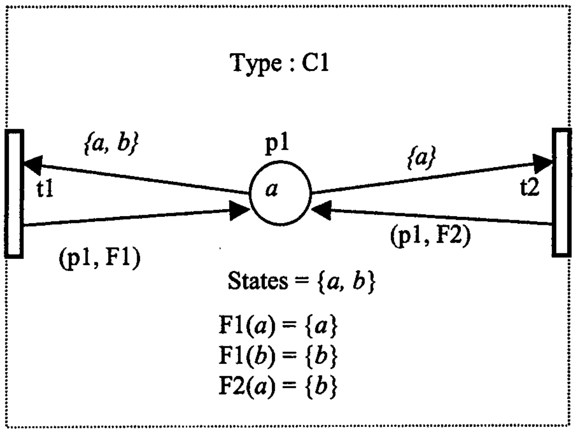


Figure 2. A Model for Object *C1*

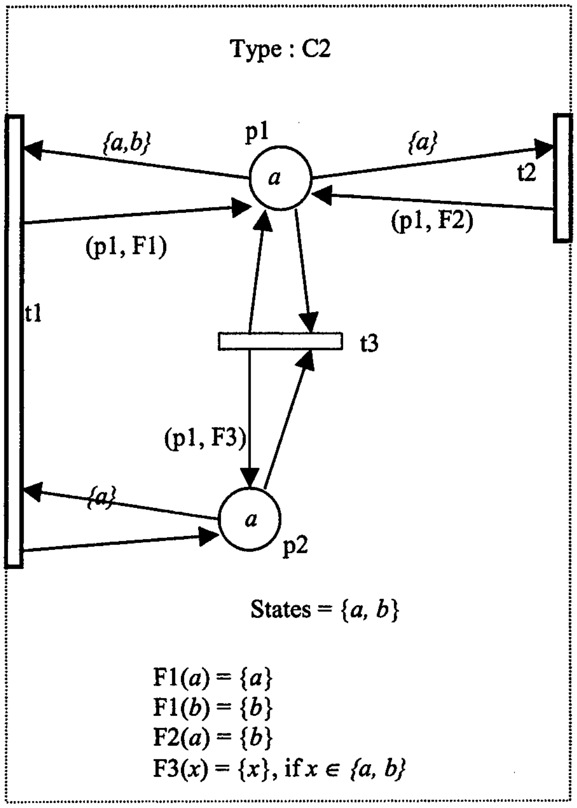


Figure 3. A Model for Subclass Object C2 (Incomplete)

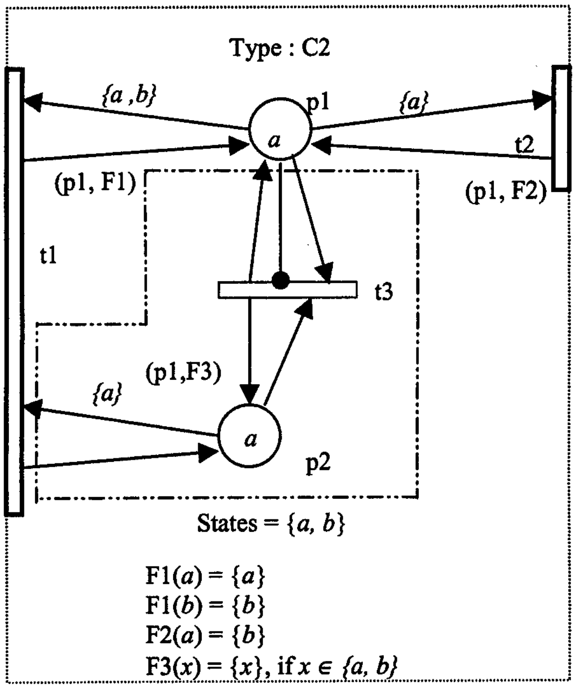


Figure 4. A Model for Subclass Object *C2* Using a Plug-in

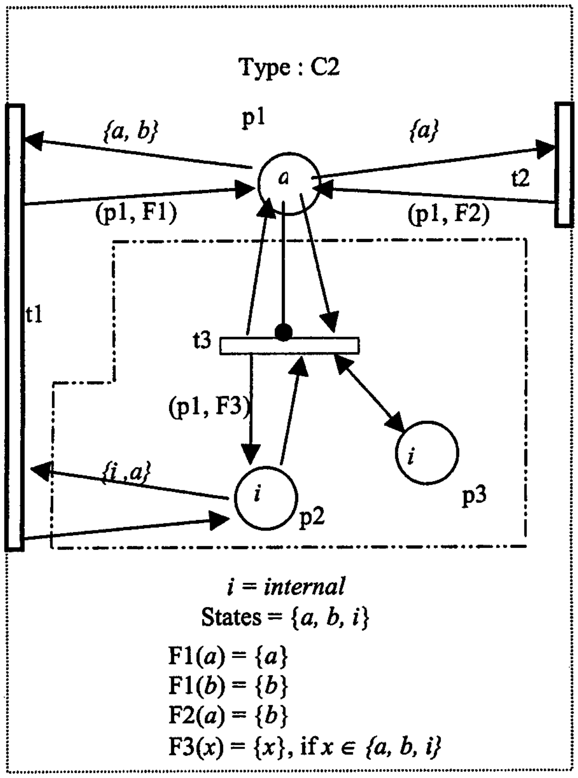


Figure 5. A Model for Object C2
Using a Switchable Plug-in

Use of Object Oriented Model for Interoperability in Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems[+]

Paul Young, Valdis Berzins, Jun Ge, Luqi

Department of Computer Science
Naval Postgraduate School
Monterey, California 93943, USA

Email: {young, berzins, gejun, luqi}@cs.nps.navy.mil

ABSTRACT

One of the major concerns in the study of software interoperability is the inconsistent representation of the same real world entity in various legacy software products. This paper proposes an object-oriented model to provide the architecture to consolidate two legacy schemas in order that corresponding systems may share attributes and methods through use of an automated translator. A Federation Interoperability Object Model (FIOM) is built to capture the information and operations shared between different systems. An automatic translator generator is discussed that utilizes the model to resolve data representation and operation implementation differences between heterogeneous distributed systems.

Key words: interoperability, object-oriented model, federation interoperability object model, wrapper

1. INTRODUCTION

In contemporary object-oriented modeling, an object is a software representation of some real-world entity in the problem domain. An object has identity (i.e., it can be distinguished from other objects by a unique identifier of some kind), state (data associated with it), and behavior (things you can do to the object or that it can do to other objects). In the Unified Modeling Language (UML) these characteristics are captured in the name, attributes, and operations of the object, respectively. UML distinguishes an individual object from a set of objects that share the same attributes, operations, relationships, and semantics-termed a *class* in the UML. [BRJ99]

This view of objects and classes has proven valuable in the development of countless systems in various problem domains encompassing all degrees of size and complexity. However, one common characteristic of the majority of these object-oriented developments is that a development team that shared common objectives and had a common view of the real-world entities being modeled produced them. Often, the developments also involved a common architecture implemented on a common target platform, using the same implementation language and operating system. As a result a single method of representation of an entity's name, attributes, and operations is the norm. Even on heterogeneous implementations by the same development team, consistency in the names, attributes and operations used for the same real-world entity is likely across the various elements of the architecture. Therefore, capturing the representation of these properties has not been an issue. The software representation of the real-world entity should have the same name, attributes, and operations across all elements of the architecture if the development team enforces consistency.

This is not necessarily the case when independently developed, heterogeneous systems are targeted for integration and interoperation. The different perspectives of the real-world entity being modeled by independent development teams will most likely result in the use of different class names as well as differences in the number, definition, and representation of attributes and operations for the same real-world entity implemented on two or more different systems. It is the same situation for non-object-oriented fashioned systems. These differences in representation of the same real-world entity on different systems must be reconciled if the systems are to interoperate.

This paper proposes an object-oriented model for defining the information and operations shared between systems. The initial use of the model is targeted for integration of legacy systems, which generally have not been developed using the object-oriented paradigm. Defining the

[+] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA.

interoperation between systems in terms of an object model however, provides benefits in terms of the visibility and understandability of the shared information and provides a foundation for easy extension as new systems are added to an existing federation. The object model defined in this paper can be easily constructed from the external interfaces defined for most legacy systems (whether object-oriented or not).

Section 2 will introduce the Object-Oriented Model for Interoperability (OOMI) and its structure. In Section 3, an interoperability object model is defined for a specified federation of systems. Section 4 presents a overview of the use of this Federation Interoperability Object Model (FIOM) by a wrapper-based translator for enabling interoperability among legacy systems.

2. OBJECT-ORIENTED MODEL FOR INTEROPERABILITY

An extension of the contemporary object model class diagram, depicted in Figure 1, is proposed to model the different possible ways an object might be represented in a federation of independently developed heterogeneous systems. The proposed extension includes information about the different representations that an object's attributes and operations may take in different systems in the federation.

Class Name
Attributes: α, β, χ, δ, ϕ, ...
Operations: A, B, X, Δ, Φ, ...

Figure 1. Contemporary Object Model for Each System

This alternative object model includes the following extensions to the contemporary object model. First, as depicted in Figure 2, the object oriented model for interoperability (OOMI) class diagram will contain a representative of all attributes included in any defined representation of the real-world entity modeled by that class. In Figure 2 these are depicted as attributes α through ϕ. Each attribute may have multiple representations, resulting from differences in interpretation by the component system design teams. From Figure 2, each of attributes α through ϕ has n representations, labeled α_{R1} through α_{Rn} for attribute α, and similarly for each of the other attributes. A standard representation for each attribute is also included, labeled α_{STD} for attribute α in Figure 2. The standard representation is chosen by the interoperability designer as an intermediate representation to be used during translation.

For each attribute representation, the interoperability object model class diagram will contain information used in establishing that the different representations refer to a common characteristic of the real-world entity being described. This includes information about both the syntax of the attribute (attribute type, structure, size, etc.) and the semantics of the attribute (attribute role, description, etc.). This information is depicted for attribute α representation 1 in Figure 2 as α_{R1} *Syntax* and α_{R1} *Semantics*, respectively.

In addition, the model will contain one or more translations required to convert between different representations of that attribute. These translations can be defined on a pair-wise basis for all possible representations- requiring *n(n-1)* translations for *n* different representations. Alternatively, they can be defined using the standard representation as an intermediate representation and translation performed in two steps (representation *1* to standard to representation *2*), requiring *2n* translations. The two-step translation method is depicted in Figure 2, with translation α_{R1}*ToSTD()* defined to translate an instance of attribute α from representation *1* to the standard representation, and translation *STDTo*α_{R2}*()* defined to translate an instance of the standard representation of attribute α to representation *2*.

Similarly, the interoperability object model class diagram extends the contemporary object model class diagram to include information about different possible implementations for each operation. Implementation differences may include differences in operation and parameter naming, differences in the number and type of parameters invoked by the operations, and differences in the internal algorithms used by each operation. As long as the dynamic behavior of the two implementations is equivalent for the same input and output conditions, they can be used interchangeably. Thus, the OOMI class diagram includes information necessary to determine if different implementations of an operation are inter-accessible. This includes information about both the syntax of the operation (naming, parameters, etc.) and the semantics of the operation (operation role, behavior, description, etc.). In addition, for each operation, the model will contain one or more translations required to account for operation name and parameter variations found in different operation implementations. Figure 3 illustrates the operation extension provided in the OOMI class diagram.

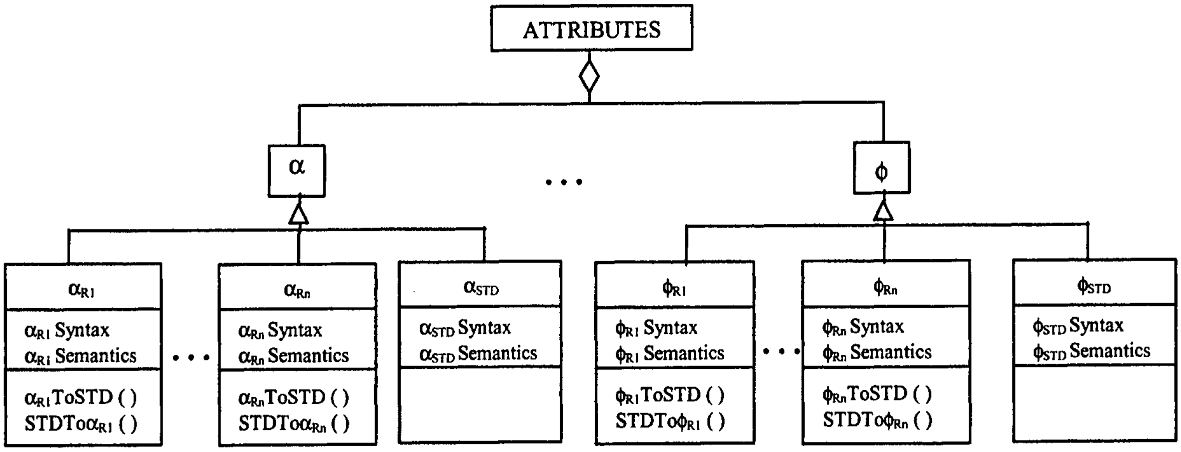


Figure 2. OOMI Class Diagram Attribute Extension

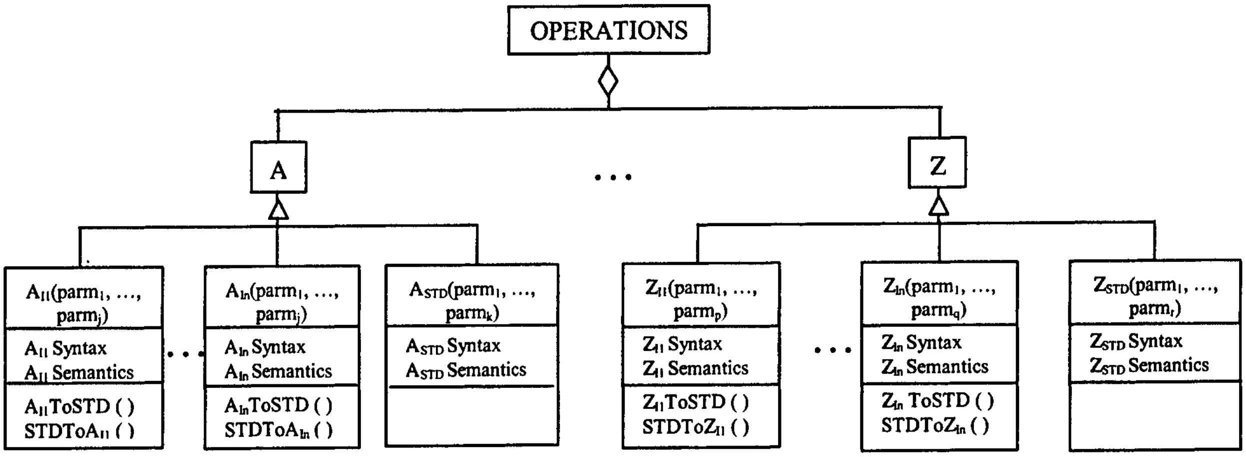


Figure 3. OOMI Class Diagram Operation Extension

From Figure 3, it can be seen that the depicted class diagram contains operations A through Z and that each operation has a number of different implementations. For example, operation Z has implementations Z_{I1} through Z_{In}, each with a potentially different set of parameters. For each operation, the interoperability designer defines a standard implementation for that operation which is used as an intermediate representation during translation. For each implementation syntactic and semantic information is provided in order to establish a correspondence with other operation implementations that are equivalent- for example *Z_{In} Syntax* and *Z_{In} Semantics* for operation Z implementation n. Finally, translations *Z_{In}ToSTD()* and *STDToZ_{In}()* are used to translate operation and parameter names from operation Z implementation n to the standard representation for operation Z's name and parameters, and vice versa.

In addition to having different representations for the same attribute or different implementations for an operation, heterogeneous object designers may provide different numbers and types of attributes and operations for the same real-world entity. One representation of that real-world entity might include attributes and operations that another representation omits. Because of this difference, a mechanism must be provided to capture the attributes and operations present in the various representations of the entity. This is provided through the addition of a Class Structure property to the interoperability object model class diagram.

Figure 4 depicts the OOMI class structure property for an example class. A representation of this class is found in the external interface of a number of systems, as specified by the *ClassA* through *ClassX* class diagrams that comprise the aggregate Class Structure property. For each representation, a list of the attributes and operations included in that representation is included. In addition, the system of origin of the class and whether the class is exported (*ProducerClass*) or imported (*ConsumerClass*) by the system is also included in the class's attribute property. As indicated in Figure 4, *ClassA* contains attributes $Aattr_1$ through *$Aattr_n$* and operations Aop_1

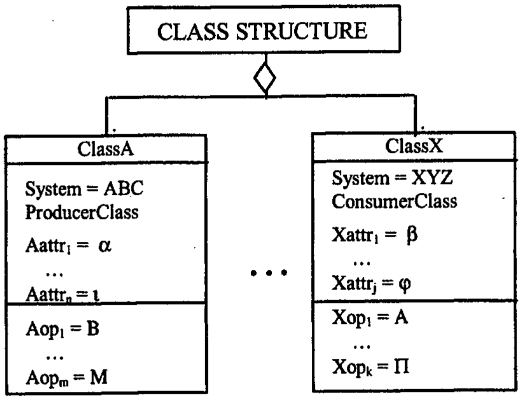


Figure 4. OOMI Class Diagram Class Structure

through Aop_m. Attribute and operation names for $Aattr_1$ through $Aattr_n$ and Aop_1 through Aop_m are the names used by system *ABC* as contained in *ABC*'s external interface. In addition to listing the attributes and operations included for each representation, the attributes and operations are identified in terms of the standard names provided in the attribute and operation properties of the class. These standard names are used together with the local names to locate the translations used to convert the attributes and operations to a different representation (to or from a standard representation).

In summary, the Object-Oriented Model for Interoperability is an extension of the contemporary object model, augmenting the contemporary model class diagram with a Class Structure property and extending the Attribute and Operation properties to capture the different representations possible for those properties in a federation of autonomous heterogeneous systems. The model is extensible in that adding new representations for an attribute or operation or for a class merely adds a class to the existing properties while preserving the existing representations. The model increases the level of abstraction dealt with by the interoperability engineer by enabling him to think in terms of the real-world entities participating in the interoperation between systems and not in terms of the different representations used. And finally, by capturing the information needed to represent the relationships between entity representations and the translators necessary to convert between representations, the OOMI supports automated conversion between object representations. Figure 5 provides a top-level summary of the proposed OOMI Class Diagram.

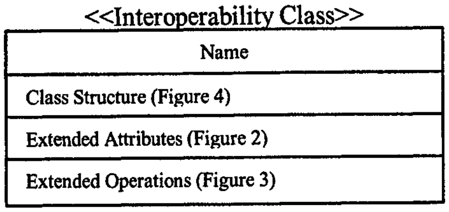

<<Interoperability Class>>
Name
Class Structure (Figure 4)
Extended Attributes (Figure 2)
Extended Operations (Figure 3)

Figure 5. OOMI Class Diagram

3. CONSTRUCTING INTEROPERABILITY OBJECT MODEL FOR FEDERATION OF HETEROGENEOUS SYSTEMS

The previously introduced Object Oriented Model for Interoperability enables information sharing and cooperative task execution among a federation of autonomously developed heterogeneous systems. Using the information contained in the OOMI class diagrams computer aid can be applied to the resolution of data representational differences between heterogeneous systems. In order to apply computer aid, a model of the real-world entities involved in the interoperation, termed a Federation Interoperability Object Model (FIOM), is constructed for the specified system federation. Construction of the FIOM is done prior to run-time by a system designer with the assistance of a specialized toolset, called the Object Oriented Model for Interoperability Integrated Development Environment (OOMI IDE).

The process of constructing a FIOM for a specified system federation essentially consists of identifying the real-world entities that reflect the shared information and tasks and capturing the different representations used by systems in the federation for that entity. Each real-world entity is represented in the FIOM as a class, termed a Federation Interoperability Class (FIC), constructed from the classes contained in the component systems' external interface.

Determination of the real-world entities that define the interoperation of a federation is not merely a matter of identifying the classes involved in the external interfaces of the systems in the federation. Because of the independently developed, heterogeneous nature of the systems in the federation, each system may have a different representation for the real-world entities involved. Thus, the classes and objects that realize the external interfaces of the component systems must be correlated to determine which representations reflect the same real-world entity. Correlation software is included as part of the OOMI IDE in order to assist the system designer by providing a small set of selected correspondences to be reviewed by domain experts.

4. AUTOMATIC WRAPPER GENERATION

System interoperability involves both the capability to exchange information between systems and the ability for joint task execution among different systems. [PIT97] Both capabilities involve one or more of the following kinds of actions:

- *Send* One system transmits a piece of information to another
- *Call* One system invokes an operation on another
- *Return* Returns a value to the caller
- *Create* Creates an object on the called system
- *Destroy* Destroys an object on the called system [BRJ99]

Information exchange is accomplished through means of a *Send* operation, where one system, the producer, exports information that another system, the consumer, imports. Information transmitted by the producer system can be in the form of an object of some class defined for the producer, or it can consist of one or more attributes of an object defined for the producer.

Joint task execution is accomplished through the use of a *Call* operation where one system, a client, invokes an operation on another, acting as a server for the requested action. In invoking an operation on a server, a client system must provide the name of the operation requested as well as any parameters required by the server to perform the operation. Required parameters can be in the form of one or more attributes, operations, or objects. In addition, in response to a client *Call* operation, a server may return a set of attributes, operations, or objects to a client via a *Return* operation. *Create* and *Destroy* actions are special instances of a system call.

When information exchange or joint task execution is performed between heterogeneous systems, the participating systems must account for differences in representation of the transmitted information. The interoperability object model constructed during the pre-runtime phase for a specified federation of component systems is used to resolve differences in representation between interoperating systems. A *translator* that serves as an intermediary between component systems accomplishes representational difference reconciliation at runtime.

The translation function is anticipated to be implemented as part of a *software wrapper* enveloping a producer or consumer system (or both) in a message-based architecture, or alternatively as part of the data store (actual or virtual) in a publish/subscribe architecture. A software wrapper is a piece of software used to alter the view provided by a component's external interface without modifying the underlying component code. Figure 6 presents an overview of the use of software wrappers and the involvement of the Federation Interoperability Object Model in the translation process.

The translations required by the wrapper-resident translator for both information exchange and joint task execution are similar. For information exchange, the source system provides the exported information in the form of a set of attributes or objects of a producer class in the native format of the producer. In order to be utilized by a consumer system, the exported information must be converted into the format expected by the destination system. For joint task execution, a client system provides an operation name and a set of parameter values to a server system in the native format of the producer. The parameters may be attributes, operations, or objects of a client class. Again, this information must be provided to the destination system in a format recognized by that system. Thus the operation name, operations, and parameter values must be converted to the server representation.

As indicated above, the translator must be capable of converting instances of a class's attributes and operations (or both attributes and operations in the form of an object of the class) from one representation to another. The information required to effect these translations is captured as part of the FIOM during federation design. As presented in Figures 2 and 3, each attribute and operation of a class representing a real-world entity defining the interoperation includes methods to enable the translation between attribute and operation representations. Then, at run time, the translator accesses the information contained in the model to effect the translation between representations.

The first action the translator must perform is to determine the class defining the real-world entity corresponding to a transmitted object, attribute, or operation. This can be accomplished through the use of a mapping developed from the FIOM that maps attribute, operation, or object representations to the class representing the corresponding real-word entity in the model. For instance, from the example provided in section 2, objects of class *ClassA* and *ConsumerX* as well as the attributes and operations for these classes would map to a real-world entity represented by prototypical class instance *RealWorldEntityA*. Once the class corresponding to the transmitted object, attribute, or operation is determined, the methods defined for each attribute and operation can be used to effect the translation between representations.

If the transmitted entity were set of attributes, such as would be the case during information exchange, then for each attribute in the set the appropriate translation method

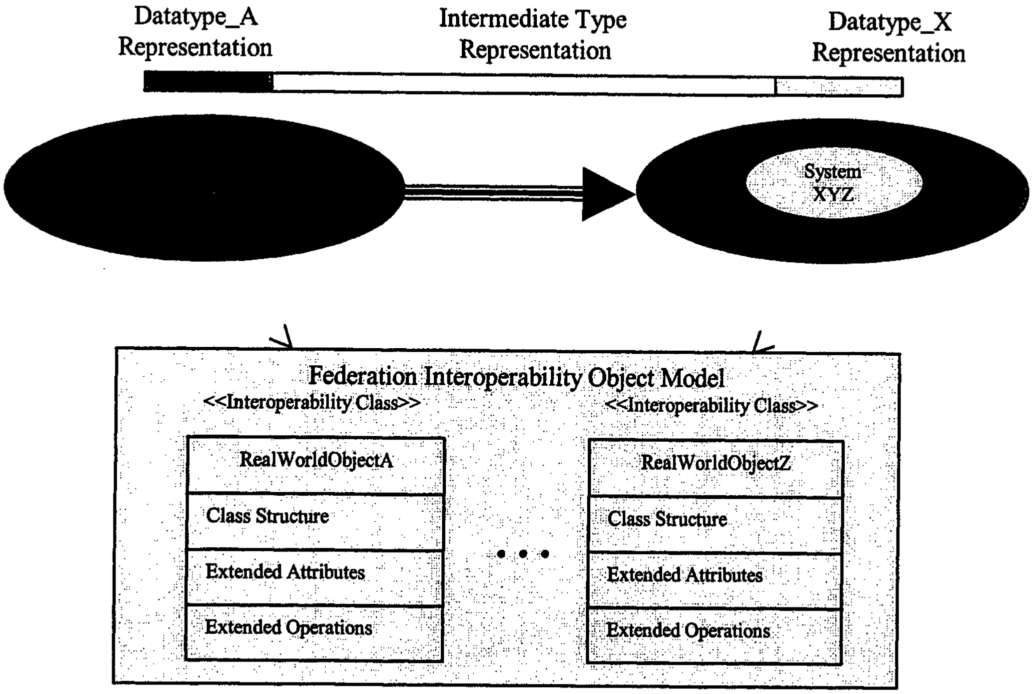


Figure 6. FIOM in Automatic Wrapper Generation

must be selected. The appropriate translation method is located by using the Class Structure property to determine the standard representation for each attribute and then finding the translations for that attribute in the Attributes property for the class representing the real-world entity. The translation provided would either be in terms of a source-to-destination or a source-to-intermediate representation conversion depending on the approach used by the system designer for the federation. In this manner the translator invokes the appropriate translation method for each attribute to convert the attribute from the source system representation to either the destination system or intermediate representation. The translated attribute set is then forwarded to the destination system for appropriate disposition. If an intermediate representation is used in the translation process, this process is repeated by the destination system to convert from the intermediate to destination system representation.

For instance, continuing our example from section 2, suppose System *ABC* were to transmit the attributes *Aattr*$_1$ and *Aattr*$_2$ from class *ClassA* to System *XYZ*. Then presuming that the representation used for System *ABC* is not useable by System *XYZ*, *Aattr*$_1$ and *Aattr*$_2$ must be translated to a form useable by System XYZ. For our example a wrapper-based translator on Systems *ABC* and *XYZ* will conduct the translation with the translation performed in two steps using an intermediate representation of the real-world entity's attributes.

As depicted in Figure 7 below, the System *ABC* wrapper would intercept the transmitted attributes from System *ABC*. Then, using the mapping outlined above, the wrapper-based translator would first determine that the intercepted attributes were of class *ClassA* that corresponds to class *RealWorldEntityA* representing the real-world entity participating in the interoperation. Then, for each attribute, the appropriate translation method must be determined. This translation method can be determined from the Attributes property, given the standard representation for the attribute. From *RealWorldEntityA* 's Class Structure property (see Figure 4), it is determined that *ClassA* attribute Aattr$_1$ corresponds to *RealWorldEntityA* 's type Attribute_α and Aattr$_2$ corresponds to type *Attribute_*β. The appropriate translation method is then selected- *Attribute_*α translation *1* (*Aattr*$_1$*ToSTD()*) for *ProducerA* attribute *Aattr*$_1$ and *Attribute_*β translation *1* (*Aattr*$_2$*ToSTD()*) for *ProducerA* attribute *Aattr*$_2$. The translator would apply these translation methods to each attribute as appropriate and forward the resultant intermediate representation to System *XYZ*.

The System *XYZ* wrapper would intercept the incoming transmission and repeat the process outlined above to convert the attributes from their intermediate representation to the *ConsumerX* representation as depicted in Figure 7. The resultant translated attributes would then be forwarded to System *XYZ* for disposition.

If the transmitted entity is an operation with a set of parameters, such as would be the case during joint task execution, then the translator must enable conversion of both the operation name and parameters and translation methods for both operation name and parameter set must be selected. The appropriate translation method for converting the operation name is located by using the Class Structure property to determine the standard representation for the operation name and then finding the translations for that operation name in the Operations property for the class representing the real-world entity. The translation provided would either be in terms of a source-to-destination or a source-to-intermediate representation conversion depending on the approach used by the system designer for the federation. The translator would then invoke the appropriate translation method for the operation to convert the operation name from the source system representation to either the destination system or intermediate representation.

Operation parameters can either be attributes, objects, operations, or their combinations. For attribute parameters, translation of each attribute is conducted as described in the attribute translation process above. Translation of object parameters will be discussed in the next paragraph. Operation parameter translation would involve both operation name and parameter translation as described above. The translated operation name and parameter list is then forwarded to the destination system for appropriate disposition. As described above for attribute translation, if an intermediate representation is used in the translation process, this process is repeated by the destination system to convert from the intermediate to destination system representation.

As an example of operation translation, suppose System *ABC* wanted to invoke an operation on System *XYZ* that corresponded to System *ABC* operation Aop_1. Such a situation might arise if operation Aop_1 involved a query of system *ABC*'s database and an equivalent operation to find similar information in System *XYZ*'s database was desired. In order for System *ABC* to perform such a task, an equivalent implementation of operation Aop_1 must exist on System *XYZ* and any differences in representation between Aop_1's name and parameters must be resolved for System *XYZ* to be able to execute the operation call. Resolution of representational differences is accomplished by wrapper-based translators on Systems *ABC* and *XYZ* using an intermediate representation of the real-world entity's operations and parameters in a similar manner as was previously done for attributes.

As depicted in Figure 8 below, the System *ABC* wrapper would intercept the transmitted operation from System *ABC*. Then, using the mapping outlined above, the wrapper-based translator would first determine that the intercepted operations were of class *ClassA* that corresponds to class *RealWorldEntityA* representing the real-world entity participating in the interoperation. Then, for each operation name and parameter, the appropriate translation method must be determined. For the operation name, the translation method can be determined from the Operations property, given the standard representation for the operation name. From *RealWorldEntityA* 's Class Structure property (see Figure 4), it is determined that *ClassA* operation Aop_1 corresponds to *RealWorldEntit yA* *Operation_B* and operation Aop_2 corresponds to *Operation_A*. The appropriate translation method is then selected-*Operation_B* translation *1* (*Aop_1ToSTD()*) for *ProducerA* operation Aop_1 and *Operation_A* translation *1* (*Aop_2_To_STD()*) for *ProducerA* operation Aop_2.

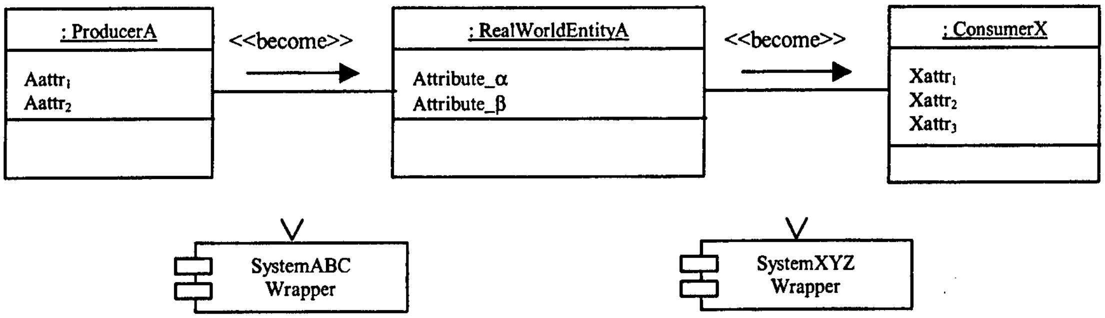


Figure 7. Mapping Translation to Wrapper Architecture

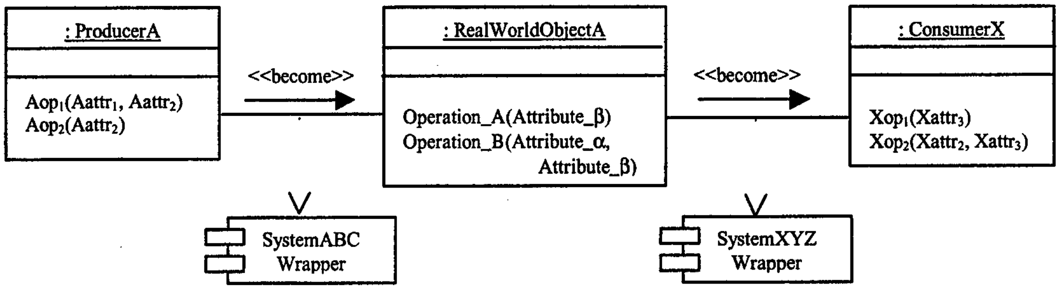


Figure 8. Wrapper-based Translator

In addition to translating the operation name, differences in representation of the operation's parameters must also be resolved. For our example, converting parameter representations would be accomplished in the same manner as done previously for converting attribute representations. The translator would apply these translation methods to each operation name and parameter as appropriate and forward the resultant intermediate representation for the operation to System *XYZ*.

The System *XYZ* wrapper would intercept the incoming transmission and repeat the process outlined above to convert the operation names and parameters from their intermediate representation to the *ConsumerX* representation as depicted in Figure 8. The resultant translated operations would then be forwarded to System *XYZ* for disposition.

Translation of object representations involves a combination of the procedures for attribute and operation conversion outlined above. First though, a correspondence between the source and destination object's class attributes and operations must be determined from the Class Structure property. If an intermediate representation is used to effect the translation, the correspondence between the source and intermediate representation of the object's class must be determined. Once the attribute and operation correspondence is established between representations, the methods for attribute and operation translation outlined above are used to convert between representations. Again, for translations involving an intermediate representation, the process must be repeated by the destination system to convert from the intermediate to destination system representations.

5. CONCLUSIONS

An Object-Oriented Model for Interoperability (OOMI) is proposed in this paper to solve the data and operation inconsistency problem in legacy systems. A Federation Interoperability Object Model (FIOM) is defined for a specific federation of systems designated for interoperation. The data and operations to be shared between systems are captured in a number of Federation Interoperability Classes (FICs) used to define the interoperation between legacy systems. Software wrappers are generated according to the FIOM that enable automated translation between different data representations and operation implementations.

At this stage, XML-based message translation is being studied for implementation of the proposed model. The capability provided by the XML family of tools coincides nicely with the requirement for data and operation representation capture and translation.

Some important issues, such as security, real-time, etc., are not discussed in this paper. However, the structure of the semantic and/or syntactic information integrated in the OOMI preserves the capability of being extended to address such concerns.

REFERENCES

[BRJ99] Booch,G., Rumbaugh, J., Jacobsoñ, I., *The Unified Modeling Language Us er Guide*, Addison-Wesley Longman, Inc., Redding, MA, 1998.

[Pit97] Pitoura, E., "Providing Database Interoperability through Object-Oriented Language Constructs", Journal of Systems Integration, Volume 7, No. 2, August 1997, pp. 99-126.

Intelligent Software Decoys

James Bret Michael and **Richard D. Riehle**
Department of Computer Science, Naval Postgraduate School
833 Dyer Road, Monterey, California 93943-5118
bmichael@cs.nps.navy.mil, rdriehle@nps.navy.mil

Abstract

We introduce an abstraction known as an intelligent software decoy for protecting objects within a component-based architecture from egregious and malicious use by mobile agents. If an agent misuses or tries to circumvent the published interface specification of an object, then the object switches from its nominal operating mode to a deception mode. While serving as a decoy, an object attempts to both deceive the agent into concluding that its violation of the interface specification has been successful and assess the nature of the violation. The interface specification is treated as a contract consisting of preconditions, postconditions, and a class invariant. Failure of a precondition triggers the transition between modes. An intelligent software decoy is adaptable, autarkic, polymorphic, and self-replicating. The decoy disguises and defends itself by modifying its contract at run-time through the use of both polymorphism and late binding. The nature and extent of any change to an object is governed by its class invariant.

Keywords: Agent, authentication, buffer overflow, component-based software architecture, contract, deception, decoy, distributed systems, encryption, object, polymorphism, security, software

1. INTRODUCTION

Suppose that there exists a distributed system of thousands of sensors, comprising part of an intelligence-gathering information system, in which each sensor is field-programmable via software hooks. An intelligence analyst could broadcast messages to the sensors instructing them to either activate or deactivate themselves, change their mode of operation (e.g., from filtering to no filtering of sensor inputs), or install and execute new software (e.g., for controlling sensing of phenomena or encrypting communication with the data-collectors). In addition, the analyst could query the status of the sensors to assess the organization's level of readiness for tracking an enemy's movement of troops and weapons.

On arriving at the software interface of a sensor, a mobile agent interacts with the sensor to reach the goal assigned to the agent by its owner. However, if the mobile agent is poorly designed, its flaws may lead the agent to try to interact with the sensor software in a way that was not intended by the creator of the agent; this is an example of an egregious but non-malicious use of the object. This could include unintentionally triggering a change in the operating mode of the sensor. If the software agent is malicious, it may try to sabotage the sensor. For example, it might try to alter the sensor software so that the movement of enemy forces will not be detected or reported by the sensor. If the software is written in Java, the agent might try to change the behavior of one of the objects or classes. In early versions of the Java Virtual Machine (JVM), such an attack was quite easy to effect due to the fact that a rogue process could insert its own class definition using the same name as the original predefined Java class [13].

The intelligence-gathering example illustrates the need for permitting mobile agents to modify the software-controlled behavior of a distributed system or a subset of the objects that comprise the system. On the other side of the coin, the object needs to be protected from egregious or malicious acts by an agent to misuse the object or modify the object in an unintended or unauthorized manner. By egregious, we mean an unintentional or non-malicious use or modification of the object's interface or behavior, while malicious refers to an attack on the object.

One approach to protecting the intelligence-gathering system is to both encrypt the messages and authenticate the mobile agents to the software objects. However, McHugh and Michael have identified some of the challenges in managing cryptographic keys in distributed systems, especially when group membership (e.g., subgroups of the sensors) changes frequently [14]. Moreover, an authenticated mobile agent may have been compromised, or its creator, who at one time was trustworthy, may no longer be so. In summary, encryption and authentication do not address the issue of discovering and responding to the goals or actions of mobile agents.

Another example of an approach to protecting distributed systems from mobile agents is to require that the agents only interact with software objects via a formal interface specification known as a software contract, as introduced in Meyer's design-by-contract model [16]. However, an agent might try to bypass the contract to modify the behavior of the targeted object. Thus, precondition assertions for controlling access to the object may only be effective at thwarting the actions of non-malicious agents, that is, agents whose flawed design induces unintended interactions with objects through the interfaces to these objects. This is known in the epigrammatic world as "Locks are intended to keep honest people honest."

We introduce an abstraction known as an intelligent software decoy for protecting objects within a component-based architecture from egregious and malicious use by mobile agents. If an agent misuses or tries to circumvent the published interface specification of an object, then the object switches from its nominal operating mode to a deception mode. While serving as a decoy, an object attempts to both deceive the agent into concluding that its violation of the interface specification has been successful and assess the nature of the violation. The interface specification is treated as a contract consisting of preconditions, postconditions, and a class invariant. Failure of a precondition triggers the transition between modes. An intelligent software decoy is adaptable, autarkic, polymorphic, and self-replicating. The decoy disguises and defends itself by modifying its contract at run-time through the use of both polymorphism and late binding. The nature and extent of any change to an object is governed by its class invariant.

2. SOFTWARE DECOYS

A decoy is intended to deceive something or someone into believing it is the object it advertises itself to be. Therefore, the creator of a decoy must actualize the decoy as much as possible to complete the deception. The more the external observer is deceived, the better the decoy is performing its role. Daniel and Herbig define deception as the "deliberate misrepresentation of reality done to gain a competitive advantage" [5].

When a duck hunter deploys decoys on a lake, those decoys are painted to resemble the species of duck being pursued. If the decoys can be made to move about, the deception may be more effective: the real ducks will think that the decoys are also real since the decoys appear to be paddling through the water. In this case, the effectiveness of the decoys need only be good enough so as to draw the real ducks within shotgun range.

An intelligent software decoy has some of the same properties as the physical decoy. It certainly has the same objective: deception. If the decoy is intelligent, it can continually deceive the target of the deception into action that accomplishes several goals. In the case of an attack or the deployment of countermeasures executed by an attacker, one of the goals of the owner of the decoy is to protect the actual entity being shielded from attack and anti-decoy countermeasures.

Another goal, in the context of an attack, is to ensure that every attack reveals the presence of an attacker. In this way, the decoy can use its own intelligence to deploy more decoys and to alert other objects that an attack signature has been identified. As more decoys are deployed, their creator can also alter their own characteristics so that the decoys appear to be different from the one originally attacked.

In an ideal situation, the decoys will be able to adopt a chameleon-like character that allows them to appear to be different as other decoys and attackers change form. In the context of software decoys, this model of decoys raises the concept of intelligent agents to a new level of sophistication. It requires that both the interfaces and the objects be polymorphic, that is, the contract for each object must be polymorphic. Consequently, any message to a decoy can be encrypted, but the decoy will have its own knowledge of the encryption scheme based on the parameters of the polymorphic message. Successful execution of the decoy will require satisfying the precondition, the invariant, and the postcondition. Since the postcondition is internal to the object, it is not easily compromised even with dynamic patching schemes.

3. PRIOR RESEARCH

The general notion of a software decoy is not new. For example, the term "decoy" has been used in the context of reasoning with incomplete information in multiagent systems. According to Zlotkin and Rosenshein [27],

> One obvious way in which uncertainty can be exploited can be in misrepresenting an agent's true goal. In a task oriented [*sic*] domain, such misrepresentation might involve hiding tasks, or creating false tasks (phantoms, or decoys), all with the intent of improving one's negotiating position. The process of reaching an agreement generally depends on agents declaring their individual task sets, and then negotiating over the global set of declared tasks. By declaring one's task set falsely, one can in principle (under certain circumstances), change the negotiation outcome to one's benefit.

This earlier research indirectly addresses the Byzantine Generals problem [12] in that there was an attempt to construct incentive-compatible negotiation mechanisms such that "no agent designer [*sic*] will have any reason to do anything but make his agent declare his true goal in a negotiation." In contrast to the work of Zlotkin and Rosenshein, in which interaction between agents was investigated, we explore the use of software decoys in the context of the interaction between agents and software components.

Turing introduced the "imitation game" [25], now known as the Turing test, for testing the intelligent behavior of software. The participants in the test consist of a computer, a human, and an interrogator. The goal of the interrogator, who is a human subject, is to distinguish between the computer and the human with whom he or she carries on a conversation. The identity of the respondent, that is the computer or human, is hidden from the interrogator. The measure of intelligent behavior of the software system is the percentage of time that the interrogator cannot correctly distinguish between the response of the computer, which simulates a human response, and that of the person typing responses. Thus, the game is one of deception: programming a machine to deceive, via impersonation, a human into believing that the machine he or she is conversing with is also a human being.

In contrast to the approach taken by Turing to test for computer intelligence, Goldberg [8] attempts to address questions of intelligent reasoning by computers by arguing that a computer cannot deceive itself. His argument relies on a "commonsense view of the mind," that is, that a computer cannot possess beliefs or self-knowledge, as can a human. However, Goldberg does not address the issue of whether one computer can deceive another. In our own work, we argue that it is possible for a software component to deceive an agent by creating a deception based on either direct inspection of the internal state of the other agent, or alternatively, assessing the intentions of the agent by monitoring the agent's behavior. In addition, we subscribe to the theory posed by Hirstein that self-deception can be due to conflicts other than between beliefs, namely, a "conflict between two representations, a 'conceptual one' and an 'analog' one" [9]. Our conception of a decoy is one in which a decoy, agent, or other type of software can itself possess conflicting representations.

In [6], examples are presented of the use of deception in military campaigns dating back thousands of years. In [4], Cohen presents a classification of defenses for information systems, in which one of those defenses is deception:

> *Defence 98: deceptions.* Typical deceptions include concealment, camouflage, false and planted information, reuses [*sic*], displays, demonstrations, feints, lies, and insight (Dunnigan, 1995). Examples include facades used to misdirect attackers as to the content of a system, false claims that a facility or system is watched by law enforcement authorities, and Trojan horses planted in software that is downloaded from a site. Deceptions are one of the most interesting areas of information protection, but little has been done on the specifics of the complexity of carrying out deceptions. Some work has been done on detecting imperfect deceptions.

Cohen has explored this class of defense for use in protecting computing resources in a distributed system. He refers to such protection techniques as "defensive network deceptions" [2], and has attempted to develop formal models of defensive deceptions and the types of attackers for which these deceptions are to be used. In one of these models, the attacker is characterized as an agent "who believes that information systems are vulnerable and [the attacker] has finite resources to attack" the systems. In this model, the attacker relies on intelligence reports about the information systems in order to identify and choose a specific vulnerability of the system to target, and that the attacker will not attack unless it believes that "there exists an exploitable weakness of value." In the other model Cohen presents, the attacker and defender are both assumed to believe that all systems of positive non-zero economic worth have at least one exploitable weakness.

Cohen introduces six goals for defensive network-deceptions [2]; they are to make the following:

1. Likelihood of any individual intelligence probe encountering a real vulnerability low.
2. Likelihood of any individual intelligence probe encountering a deception high.
3. Time to defeat a deception infinite.
4. Time to detect a vulnerability once a deception is encountered from a given attack location infinite.
5. Time to detect an intelligence probe against a deception very small.
6. Time to react to an intelligence probe against a deception very small.

These goals, to some extent, have been incorporated into the Deception Toolkit (DTK) [3]. Prior to the emergence of the DTK, the most widely used type of tool for defensive network deception was the honey pot, which is still used today. A honey pot is a decoy that is placed in a highly visible location within an information system so as to draw the attention of attackers. According to Cohen, honey pots have not proved to be very effective at influencing the decision making of an attacker because each honey pot "consumes such a small portion of the overall intelligence space and has little effect on altering the characteristics of the typical intelligence probe" [2].

The DTK distributes deceptions throughout the network to be protected, with the deceptions utilizing unused network-system resources. An example of a deception that can be created using the DTK is to populate the network with IP addresses masquerading as addresses of valuable system resources: the fake IP addresses and dummy resources associated with them serve as decoys. The DTK has evolved from a simple extension to honey-pot systems to incorporate techniques to both increase the size of the search space (i.e., for a real versus decoy service) and the sparseness of actual vulnerabilities. Cohen has also used the DTK as an experimental apparatus for testing strategies to improve the quality of deceptions. The strategies he lists in [2] include the following: injecting synthetic network traffic into the network, reconfiguring the deception network over time, injecting synthetic information about the organization and its constituents into the system, and using real systems rather than software sandboxes as decoys.

Moose [17], like Cohen, has tried to model deception from a systems view. He explicitly models the evolution of pairs of stimuli and responses between the defenders of a system who are using deception techniques and that of the attackers. The modeling paradigm is intended to capture deception and counter-deception scenarios, the plans of actors (i.e., defender and attacker), uncertainty associated with intelligence information, feedback loops, and the risk models of actors.

The Denial and deception Analyst Workbench (DAWS) [11] is an interactive system used by intelligence analysts to maintain denials and deceptions, in other words, cover stories. The workbench consists of a set of integrated tools, managed by an expert system. DAWS pre-processes raw intelligence data so that it can be automatically forwarded to analysts based on pattern matches on their information-needs profiles. The other tools help the user manage denials and deceptions that are perpetuated for a target audience. DAWS and DTK are similar in that they both are designed with the human in the loop.

The development of counter-deception techniques has been a very active area of research in the information theory community. For example, in the 1970s, Gilbert et al. [7] explored the use of codes to detect evidence of deception on the part of an opponent that tries to intercept or change messages between a transmitter and its intended receiver. The opponent tries to capture message streams on a channel without letting the original transmitter or the intended receiver know that the message has been captured. The typical attack scenario involves a rogue process, such as a Trojan horse, that redirects message traffic on trusted channels or via a covert channel (i.e., a channel that bypasses the information system's reference monitor). The opponent may raise the deception to an even higher level of sophistication by implementing a man-in-the-middle attack. In such an attack, the opponent captures a message, *m*, modifies the captured message, yielding m', and makes m' looks as though it has not been tampered with. The opponent impersonates the original transmitter while forwarding $m'$to the receiver that the original transmitter had intended *m* to reach.

Recent advances in information theory, such as those reported in [10, 15, 22] have produced authentication-coding schemes for detecting deception in authentication channels with single or multiple usage (i.e., without changing the key after each message is sent). The authentication codes are used to derive the lower bounds on the probability that an opponent will successfully deceive the receiver via substitution or impersonation.

Tognazzini [24] has investigated constructive uses of deception for designing human-computer interfaces. He compares the art of illusion, as practiced by magicians, to the illusions created by the designers of graphical user interfaces, that is, the virtual reality that the user of the interface perceives. Some of the techniques that he explores are misdirection, attention to detail, and the manipulation of time. He concludes his essay with a discourse on the concept of a threshold of believability (on the part of the user of a graphical user interface) and the ethics of impersonation, in the form of anthropomorphism (i.e., software agents impersonating humans).

4. A FRAMEWORK FOR INTELLIGENT SOFTWARE DECOYS

In this section we characterize the components and connections of the architecture and framework in which the intelligent software decoys reside.

Components, Named Interfaces, and Reuse

We treat intelligent software decoys as objects within components, following the usage by Szyperski of the terms "component," "object," and "interface" to describe component-based software architectures [23].

Definition (Intelligent software decoy): An object with a contract for which a violation of one or more preconditions by an agent causes the object to try to both deceive the agent into concluding that its violation of the contract has been successful and assess the nature of the violation, while enforcing all postconditions and class invariants.

The connectors between components are named interfaces. There is no requirement for the name that a decoy-equipped object advertises (i.e., the binding of its name to remote object references in the remote object table) to other components to be unique. The interface of a decoy consists of an ordered list of arguments. The arguments can be either primitive types or object classes. In the latter case, the argument supports polymorphic types. Each class is composed of its own arguments and behavior. The arguments are used to access the methods of objects within a component, either through a remote procedure call (RPC) or remote method invocation (RMI), as shown in Figure 1. All calls to procedures or invocations of methods are communicated to the object via the object's interface.

A software decoy can replicate itself, using the same name for the cloned components. Mobile agents cannot distinguish whether an object is operating in its nominal or deception mode. In order for objects to be able to distinguish amongst themselves, one could implement the architecture using a single address space operating system such as Sombrero [21], or possibly a distributed operating system that supports object-request brokers, such as StratOSphere [26].

An intelligent software decoy can change the form of its contract interface at run-time. The modification of the form of a decoy's interface is supported by polymorphism; that is, the component inherits its interface from its parent class. The modification of the interface can involve changing one or more of the following: the number of arguments, the order of arguments, or the data type or class of arguments.

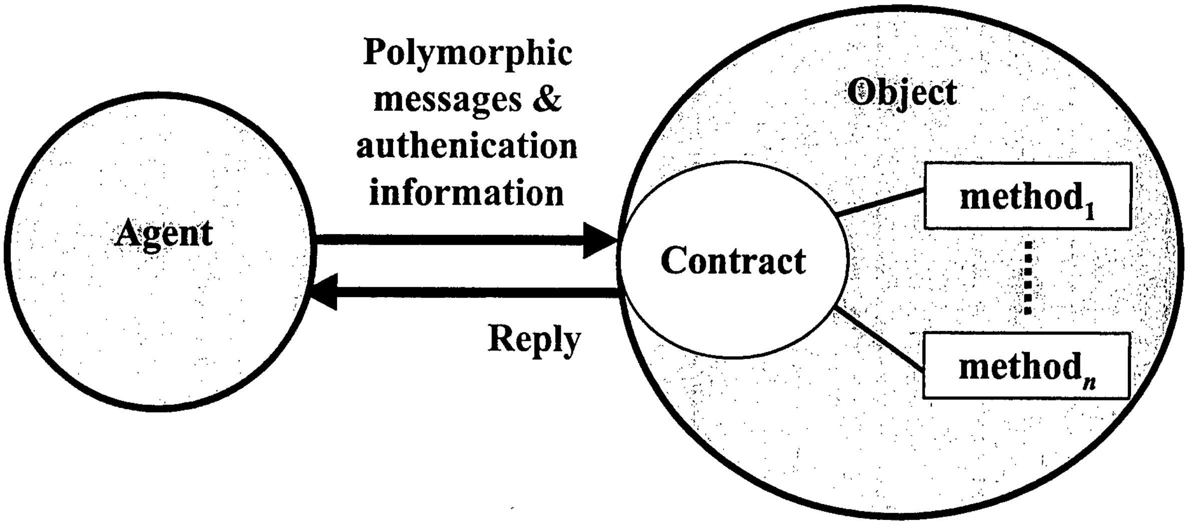


Fig. 1. Interaction between an agent and an object

The number of possible combinations of input arguments, in theory, is infinite, as is the number of class derivations. The permutation of arguments to introduce randomness into a system is not new. For example, Rothstein introduced the idea of using permuted arguments as a form of decoy in his work on message opacifiers [20].

In addition to permuting the ordering of the arguments and changing the quantity and type of arguments, randomness is injected into the interface by padding the input-argument list with one or more dummy arguments. While the total number of arguments is held constant, the position of the dummy arguments in the argument list can be changed, as can the data types of any of the arguments. The number of permutations, denoted by P, of the input-argument list for this strategy is

$$P_{m,n} = k^m \cdot (m+n)! \qquad \text{(Eq. 1)}$$

where m is the total number of dummy arguments, n is the total number of legitimate arguments, and k is the number of unique data types (both primitive and class-based) from which to assign a type to a dummy argument.

A mobile agent computes an argument list for an object it wants to access and passes that list along with authentication information to the interface of the target object. After the agent is authenticated to the object, the object verifies that the argument list that the agent passed to it is correct.

Definition (Correct agent-generated argument list): An agent-generated argument list is correct if and only if the number, ordering, and type of these arguments exactly match those of the target object's interface.

If the agent-generated argument list is correct, then the client where the object resides checks the access control list to determine whether the agent holds the permissions to access the method (e.g., execute the method locally or export the method for remote execution).

Protection of Object Behavior from Unauthorized Modification

Preconditions, postconditions, and class invariants govern the behavior of an intelligent software decoy. If the preconditions or postconditions fail during an interaction with a mobile agent, then the decoy either aborts the requested call, or raises an exception and unwinds to the caller. An alternative policy to raising exceptions is to retry the operation with a new set of data. The class invariants protect decoys from having their behavior modified in an unauthorized way. An agent cannot modify the behavior beyond the extent to which such modification is permitted by the parent class of the decoy.

Randomness can also be introduced into the design of the decoys by allowing the preconditions on the invocation of methods of a component to vary.

$$P_{m,n,q} = k^m \cdot (m+n+q)! \qquad \text{(Eq. 2)}$$

where q is the number of unique preconditions in the sample space. We do not allow for the class invariant to be permuted.

Polymorphic Types

As mentioned earlier, component interaction is based on a contract that is controlled by assertions (i.e., preconditions) as well as by a polymorphic type. The polymorphic type permits a late-binding of the message interaction. The preconditions require certain characteristics to be satisfied for each interaction to be carried forward. Preconditions are not a strong enough mechanism for all circumstances. They are particularly ineffective at guarding against mischievous action.

Polymorphic types are a little more interesting. We declare that certain parameters can have different characteristics within some accepted range of types. The types themselves may carry a set of encryption features as well as other encoding that makes them less likely to be compromised by an attacker.

An important difference in a software decoy is when the encryption error is rejected. Ordinarily, if a password fails on a routine, that routine rejects the attempt at entry. In contrast, the software decoy lets mischief proceed unnoticed by the attacker. Instead of repelling the attack, the software decoy engages it without revealing that its action is benign. This could be called the Venus flytrap model. This pleasant looking little flower lets its prey enter, enjoy the fragrance of its pollen, and encloses it for a tasty meal.

If the precondition is satisfied and the mischief is in the form of a patch, then the intelligent software decoy relies on the protection afforded by enforcement of the invariant and postcondition. Once again, if the invariant fails within the decoy, the attacker is never notified. If the postcondition fails, we apply a kind of software *jiu jitsu* within that decoy. This means we allow the attacker to believe it has been successful in overpowering the defenses while tumbling it harmlessly through the code instead of letting it forward any messages to other agents. Our approach to deception is a cross between ambiguity-increasing (A-type) deception [5], in which the decoy seeks to ensure that the "level of ambiguity always remains high enough to protect the secret of the actual operation," and misleading (M-type) deception [5], which entails reducing ambiguity by "building up the attractiveness" of a decoy, thus causing the attacker to concentrate its resources on the decoy.

Exchange of Roles

An intelligent software decoy can operate in one of two modes: nominal or deception.

Definition (Anomalous behavior of a mobile agent): An anomalous behavior of a mobile agent is one in which a request for access to a legitimate object by a mobile agent fails the test of authentication, test for correctness of the agent-generated argument list, or the check for the necessary access permissions.

Policy 1 (Transition to deception mode): If an object detects anomalous behavior during its interaction with a mobile agent, then the object transitions from nominal into deception mode, or remains in deception mode.

Policy 2 (Transition to nominal mode): An object remains in deception mode until the object or the agent terminates its interaction with the other party.

The purpose of Policy 1 is to free up the object from processing legitimate requests so that it can take on the role of a software decoy, in particular, containing the agent and gathering information about the agent. Policy 2 provides for objects to return to operating in a nominal mode.

Observation-Inference Component

The software decoy tries to determine the nature of a mobile agent's interaction with it in order to respond appropriately to the mobile agent. The software decoy records the messages passed to its interface by the mobile agent. The software decoy has a pattern recognition capability for distinguishing between whether an anomalous behavior exhibited by a mobile agent is due to an error in the mobile code or an attack by that agent.

Ante Chamber: The Response Component

The role of design-by-contract [16] is critical. There can be a failure of the precondition, in which case, we must have a response policy for precondition violations. In general, failure of a precondition means the agent will not do any of its work. The policy question remains for each agent: what action is appropriate when the precondition fails? A bad precondition may originate from a benign source or may represent an attempted attack. At the very least, the decoy keeps track of such failures. Failure of the invariant or postcondition intuitively represents a higher probability of an attack on the object. In particular, the failure of the postcondition should trigger the self-modifying behavior of the decoy.

The policy for responding to a mobile agent is embedded in the software decoy. The person or organization that owns the software decoy might specify the following policies:

Policy 3 (Containment by decoy): If a mobile agent, due to a software error in its code, passes an incorrect argument list to the software decoy or the real object, then the decoy should activate its containment countermeasure, rather than actively attack the mobile agent.

Policy 4 (Counterattack by decoy): If the mobile agent intended to attack the object, then the object should not under respond by treating the interaction as being due an egregious use of the object.

Policy 3 is intended to guard against an active attack on a non-malicious mobile agent; the result of such an attack could trigger a counterattack by the mobile agent or the mobile agent's coordinating agent owned by the same enterprise—a form of "friendly fire" due to an error in assessing the true nature of the violation of the contract. In contrast, Policy 4 dictates that the containment should include countermeasures that involve an active attack against not only the mobile agent, but also the applet that generated the agent, or even the process that invoked the applet.

The rules that dictate the responses of the object while it is in the deception mode are enforced within the decoy's antechamber. The antechamber serves as a waiting area for the requests initiated by an agent, that is, while the decoy assesses the nature of the contract violations and generates responses.

Testing for the failure of preconditions is not unlike acceptance testing performed by recovery blocks as part of a fault-tolerance strategy. Communicating recovery blocks (CRB), as introduced by Randell [19], provide for propagation of state recovery among recovery blocks. Likewise, the results of checking by the decoys of the preconditions for each of the chained procedure calls or method invocations (i.e., a procedure call resulting in another procedure call, or one method invocation resulting in another method invocation) are passed back to the object from which the calling procedure or method invocation originated.

5. LANGUAGE SUPPORT FOR INTELLIGENT SOFTWARE DECOYS

We believe that Eiffel is a natural choice of programming languages for implementing intelligent software decoys, at least for the purposes of initial experimentation with such decoys. In contrast to Ada95, Java, and other programming languages for which extensions have been implemented or proposed to permit the specification and use of contracts, Eiffel provides explicit support for design-by-contract in the form of built-in language constructs for specifying preconditions, postconditions, and class invariants. In addition, Eiffel's semantics provide for the checking of preconditions and postconditions at the object interface, rather than only when a method is invoked or exited.

In the example of software-controlled sensors, one could wrap the methods (e.g., `activate_sensor`) in the class named `INTELLIGENCE_GATHERING_SENSOR` with a contract as outlined in Figure 2.

```
class INTELLIGENCE_GATHERING_SENSOR
-- A sensor with an identification number and a status (on, off, unavailable)
feature
   definitions
activate_sensor(parameter list) is
-- routine for activating or reactivating a sensor
require
   preconditions
do
   operations
ensure
   postconditions
invariant
   invariants
rescue -- enter antechamber
   enter_antechamber(args)
end
```

Fig. 2. Outline of a contract for the class INTELLIGENCE_GATHERING_SENSOR

For example, a precondition could be that the sensor must already be deactivated before it can be activated, while the postcondition could be that the results of the reactivation tests performed on the sensor are not transferred to the collector in the clear (i.e., the data must be encrypted). One of the operations associated with activation of a sensor could be the generation of an encryption key. An example of an invariant is that the sensor-based data-filtering methods remain unchanged.

Moreover, Eiffel provides for inheritance of the assertions from ancestor classes by a descendant class, which is needed to preserve the integrity of the software contracts for the software decoys that are generated by a software component. However, not all Eiffel systems support the full range of the levels of run-time monitoring of assertions.

An exception will be raised when one or more of the object's assertions are violated; program control is transferred to the rescue clause. The decoy-specific Rescue clause shown in Figure 2 could be used to invoke the logic program contained in the decoy's antechamber; we are exploring the technical feasibility and efficacy of this approach. The Rescue clause, which is part of the Eiffel language, is used here to maintain the object in a safe state, in this case a decoy state, rather than terminate the current routine within INTELLIGENCE_GATHERING_SENSOR.

6. DISCUSSION

The use of intelligent software decoys within real-world systems would mark a major shift in the design of survivable systems. A decoy is not a honey pot. Instead, every object within a distributed system can be designed or wrapped with the functionality of an intelligent software decoy: there is no need to set aside a subset of all of the objects within a system as tempting bait. Further, the intelligent software decoy is founded upon the integration of formal methods (i.e., design-by-contract), deception techniques, and defensive-programming techniques. Moreover, the intelligent software decoy attempts to distinguish between egregious and malicious uses of an object's interface while also building and acting on stored and current signatures of agent-interface interaction: this is a departure from treating all violations of policy to be malicious in nature.

Intelligent software decoys can be introduced into information systems in an evolutionary manner. The owner of an information system can introduce intelligent software decoys into legacy systems without resorting to redesigning or reprogramming the existing objects in their entirety. Instead, objects in legacy systems, especially those that are necessary for the survivability of mission-critical systems (e.g., a military command-and-control system, including its infrastructure components) can be wrapped with contracts. As resources for making major modifications to the legacy systems or building new information systems become available, the owners of these systems can gradually develop reusable components containing contracts. For example, an enterprise could use Eiffel to wrap the current version of its proprietary implementation in the C programming language of the MPEG-2 compression protocol, and then completely rewrite the protocol in Eiffel with native support for contracts when the resources are available to do so.

The use of wrappers to implement contracts for commercial-off-the-shelf (COTS) software applications is also appealing because the users of such applications typically do not have access to the source code of those applications. Even if the users had access to the source code, the costs might outweigh the benefits of making changes directly to the source code because the software vendor may not support user-modified software or might change the functionality of the tool on a frequent basis. By treating an object as a black box, the user only invests in wrapping each new version of an application, rather than the potentially costly modification of the internals of the application. For example, the U.S. Department of Defense would likely benefit from the use of contract wrappers because that organization makes extensive use of COTS software applications to build information systems, some of which need to be trusted or are mission-critical.

Irrespective of whether an intelligent software decoy represents a component within the middleware or application of a system, the decoy relies on a strong foundation: the local network operating system or the distributed operating system. If the operating system cannot be trusted, then it is likely that a malicious agent will attack the weak operating system rather than directly attack the intelligent software decoys, especially in the case in which the attacker knows that the objects can operate in a deception mode. Therefore, it may be necessary to incorporate intelligent software decoys into the design of the operating system itself. For example, decoys could be introduced in an incremental manner into the Linux, Windows2000, StratOSphere, and Sombrero operating systems, starting with the objects in the kernels of the operating systems.

Furthermore, intelligent software decoys can be used to complement other approaches to protecting the components of an information system, such as the development of compilers that check for conditions that could trigger a buffer overflow. A buffer overflow is typified by the failure to specify and enforce contracts for methods that write data to buffer arrays. In essence, the calling object (or agent) is allowed to write past the bound of the target data structure, resulting in the overwriting of adjacent data structures in the same stack frame. The Return Address Defender (RAD) [1] is a patch to some existing compilers for both creating a safe area to maintain a copy of return addresses and wrapping applications with code to protect against buffer-overflow attacks. However, Chiueh and Hsu admit that RAD itself is susceptible to buffer-overflow attacks. The components of RAD and the compiler themselves could be implemented using intelligent software decoys, thus making those components more robust to buffer-overflow attacks.

7. CONCLUSION

Our approach to deception is different from that proposed in [2] in that we introduce the use of software contracts and polymorphism to create and manage intelligent software decoys. The software contracts are used to specify security policy and mediate the interaction under policy between the intelligent software decoy and agent: the postcondition and invariant place fail-safe constraints on the behavior of the decoy, thus permitting the decoy to allow the attacking mobile agent to interact with the decoy while containing the agent. The class invariant makes it impossible for the attacker to modify the behavior of a decoy, while polymorphism permits the decoy to change its appearance, in the form of preconditions, to the attacker. Moreover, the intelligent software decoys populate the entire system space; that is, every software component can switch modes at run-time—from nominal to deception mode, and vice versa—and replicate itself. In addition, the decoys can operate in an autonomous manner, due to their autarkic nature, or they can communicate their intentions to other software components to coordinate their actions to either deceive attackers or trace the source and nature of the attack.

8. FUTURE WORK

We are in the process of refining the mathematical formulation of intelligent software decoys, in addition to extending the typing of decoys, such as distinguishing between "volunteer" and "drafted" decoys. As a first step toward demonstrating the technical feasibility of using intelligent software decoys to protect objects from the effects of egregious or malicious uses of the object or its interface, we are using the Eiffel programming language to instrument objects with the functionality of decoys.

Moreover, we are designing a commitment protocol to address the issue of transitive closure for chains of method invocations, that is, methods calling other methods. At each invocation of a method, the object determines whether the preconditions are satisfied. However, with a chain of method invocations, it is necessary to evaluate the conformance to the contract at each invocation. We are designing a data structure akin to a sandbox in order to store the intermediate results generated by each invocation within the chain of methods. As an integral part of the antechamber, the commitment protocol is used to decide whether to commit or abort the transaction; the transaction consists of the entire chain of invocations of methods. We chose to test the protocol against buffer overflows stemming from the egregious use of an object's interface specification, in addition to the malicious use of software contracts. Buffer overflows have been used in many successful attacks on distributed systems and involve chains of method invocations. We also will address challenges in distinguishing between egregious and malicious attacks, such as false positives resulting from errors in reasoning about the temporal validity of the signatures of agent-object interaction. It is important that the decoy be able to distinguish between the types of agent-object interaction in order to minimize the likelihood of denying service to legitimate non-malicious agents.

In addition, we are exploring ways to apply intelligent software decoys in distributed databases in which lightweight objects perform queries on multidatabases. For instance, we are exploring how intelligent software decoys can be used in the DBMS-aglet framework proposed by Papastavrou, Samaras, and Pitoura [18]. For this case study, we would like to determine, for example, whether the aglets could create a successful denial-of-service attack by causing objects to replicate themselves as decoys. We are investigating this and other issues related to object persistence and garbage collection.

ACKNOWLEDGEMENTS

This research is supported by grants from the Naval Research Laboratory and the Naval Postgraduate School's Institutionally Funded Research Program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

We thank Bruce Allen, Terry Mayfield, John McHugh, Bertrand Meyer, Reg Meeson, Joel Pawloski, Christopher Slattery, Michael Wathen, and David Wheeler for critiquing previous versions of our conceptualization of intelligent software decoys. We also thank both the anonymous reviewers and the participants of the workshop for their comments.

REFERENCES

1. Chiueh, T.-C. and Hsu, F.-H. RAD: a compile-time solution to buffer overflow attacks. In *Proc. Twenty-first Internat. Conf. on Distributed Computing Systems*, IEEE (Phoenix, Ariz., Apr. 2001), 409-417.
2. Cohen, F. A mathematical model of simple defensive network deceptions. *Computers & Security* 19, 6 (2000), 520-528.
3. Cohen, F. A note on the role of deception in information protection. *Computers & Security* 17, 6 (1998), 483-506.
4. Cohen, F. Information system defences: a preliminary classification scheme. *Computers & Security* 16, 2 (1997), 94-114.
5. Daniel, D. C. and Herbig, K. L. Propositions on military deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 3-30.
6. Dunnigan, J. F. and Nofi, A. A. *Victory and Deceit.* William Morrow & Co., New York, fifth ed., 1995.
7. Gilbert, E. N., MacWilliams, F. J., and Sloane, N. J. A. Codes which detect deceptions. *Bell Syst. Tech. Jour.* 53, 3 (1974), 405-424.
8. Goldberg, S. C. The very idea of computer self-knowledge and self-deception. *Minds and Machines* 7, 4 (Nov. 1997), 515-529.

9. Hirstein, W. Self-deception and confabulation. *Jour. Philosophy of Science* 67, 3 (Suppl. S, Sept. 2000), S418-S429.
10. Johansson, T. Lower bounds on the probability of deception in authentication with arbitration. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1573-1585.
11. Kisiel, K. W., Rosenberg, B. F., and Townsend, R. E. DAWS: Denial and deception analyst workstation. In *Proc. Internat. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, vol. II, IEEE (Tullahoma, Tenn., June 1989), 640-644.
12. Lamport, L, Shostak, R, and Pease, M. Byzantine Generals problem. *ACM Trans. Programming Languages and Systems* 4, 3 (1982), 382-401.
13. McGraw, G. and Felton, E. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, New York, 1996.
14. McHugh, J. and Michael, J. B. Secure group management in large distributed systems: What is a group and what does it do? In *Proc.New Security Paradigms Workshop*, ACM (Caledon Hills, Ont., Sept. 1999), 80-85.
15. Meijer, A. R. Deception in authentication channels with multiple usage. In *Proc. IEEE South African Symp. on Communications and Signal Processing*, IEEE (Matieland, South Africa, Oct. 1994), 60-67.
16. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, N.J., 1998.
17. Moose, P. H. A systems view of deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 136-150.
18. Papastavrou, S., Samaras, G., and Pitoura, E. Mobile agents for world wide web distributed database access. *IEEE Trans. Knowledge and Data Engin.* 12, 5 (Sept.-Oct. 2000), 802-820.
19. Randell, B. System structure for software fault tolerance. *IEEE Trans. Software Engin.* 1, 2 (June 1975), 220-232.
20. Rothstein, J. Parallel processable cryptographic methods with unbounded practical security. In *Proc. Internat. Symp. on Inf. Theory*, IEEE (Ithaca, N.Y., Oct. 1977), 43.
21. Skousen, A. and Miller, D. The Sombrero single address space operating system prototype: A testbed for evaluating distributed persistent system concepts and implementation. In *Proc. Internat. Conf. on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, (Las Vegas, Nevada, June 2000), 557-563.
22. Smeets, B. Bounds on the probability of deception in multiple authentication. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1586-1591.
23. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1999.
24. Tognazzini, B. Principles, techniques, and ethics of stage magic and their application to human interface design. In *Proc. Conf. on Human Factors in Computing Systems*, ACM (Amsterdam, Neth., Apr. 1993), 355-362.
25. Turing, A. M. Computing machinery and intelligence. *Mind* 59, 236 (Oct. 1950), 433-460.
26. Wu, D., Agrawal, D., and El Abbadi, A. StratOSphere: mobile processing of distributed objects in Java. In *Proc. Fourth Annual Internat. Conf. on Mobile Computing and Networking*, ACM (Dallas, Tex., Oct. 1998), 121-132.
27. Zlotkin, G. and Rosenschein, J. S. Mechanism design for automated negotiation, and its application to task oriented domains. *Jour. Artificial Intelligence* 86, 2 (Oct. 1996), 195-244.

Software Requirements Risk and Reliability

Norman F. Schneidewind
Naval Postgraduate School
Monterey, CA 93943, USA
Voice: (831) 656-2719
Fax : (831) 372-0445
nschneid@nps.navy.mil

Abstract

In order to continue to make progress in software measurement, as it pertains to reliability, we must shift the emphasis from design and code metrics to metrics that characterize the risk of making requirements changes. By doing so we can improve the quality of delivered software, because defects related to problems in requirements specifications will be identified early in the life cycle. We developed an approach for identifying requirements change risk factors as predictors of reliability problems. Our case example consists of twenty-four Space Shuttle change requests, nineteen risk factors, and the associated failures and software metrics. The approach can be generalized to other applications with numerical results that would vary according to application.

Keywords: software requirements analysis, reliability risk, software metrics.

1. INTRODUCTION

While software design and code metrics have enjoyed some success as predictors of software quality attributes such as reliability [5, 6, 7, 8, 11, 13, 14], the measurement field is stuck at this level of achievement. If measurement is to advance to a higher level, we must shift our attention to the front-end of the development process, because it is during system conceptualization that errors in specifying requirements are inserted into the process and adversely affect our ability to develop the software. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors propagate through later phases of development. These errors may result in significant risks associated with implementing the requirements. For example, reliability risk (i.e., risk of faults and failures induced by changes in requirements) may be incurred by deficiencies in the process (e.g., lack of precision in requirements). Although requirements may be specified correctly in terms of meeting user expectations, there could be significant risks associated with their implementation. For example, correctly implementing user requirements could lead to excessive system size and complexity with adverse effects on reliability or there could be a demand for project resources that exceeds the available funds, time, and personnel skills. Interestingly, there has been considerable discussion of project risk (e.g., the consequences of cost overrun and schedule slippage) in the literature [1] but not a corresponding attention to reliability risk.

Risk in the Webster's New Universal Unabridged Dictionary is defined as "the chance of injury; damage, or loss" [21]. Some authors have extended the dictionary definition as follows: "Risk Exposure=Probability of an Unsatisfactory Outcome*Loss if the Outcome is Unsatisfactory" [1]. Such a definition is frequently applied to the risks in managing software projects such as budget and schedule slippage. In contrast, our application of the dictionary definition pertains to the risk of executing the software of a system where there is the chance of injury (e.g., crew injury or fatality), damage (e.g., destruction of the vehicle), or loss (e.g., loss of the mission) if a serious software failure occurs during a mission. We use risk factors to indicate the degree of risk associated with such an occurrence.

The generation of requirements is not a one-time activity. Indeed, changes to requirements can occur during maintenance. When new software is developed or existing software is changed in response to new and changed requirements, respectively, there is the potential to incur reliability risk. Therefore, in assessing the effects of requirements on reliability, we should deal with changes in requirements throughout the life cycle.

In addition to the relationship between requirements and reliability, there are the intermediate relationships between requirements and software metrics (e.g., size, complexity) and between metrics and reliability. These relationships may interact to put the reliability of the software at risk because the requirements changes may result in increases in the size and complexity of the software that may adversely affect reliability. We studied these interactions for the Space Shuttle. For example, assume that the number of iterations of a requirements change -- the "mod level" -- is inversely related to reliability. That is, if many revisions of a requirement are necessary before it is approved, this is indicative of a requirement that is hard to understand and implement safely -- a risk that directly affects reliability. At the same time, this complex requirement will affect the size and complexity of the code that will, in turn, have deleterious effects on reliability.

2. OBJECTIVES

Given the lack of emphasis in measurement research on the critical role of requirements, we were motivated to investigate the following issues:

- What is the relationship between requirements attributes and reliability? That is, are there requirements attributes that are strongly related to the occurrence of defects and failures in the software?
- What is the relationship between requirements attributes and software attributes like complexity and size? That is, are there requirements attributes that are strongly related to the complexity and size of software?
- Is it feasible to use requirements attributes as predictors of reliability? That is, can static requirements change attributes like the size of the change be used to predict reliability in execution (e.g., failure occurrence) of this code?
- Which requirements attributes pose the greatest risk to reliability?

2.1 Contribution

This research makes a contribution to the quantification of the above relationships, but we also point out three major problems in this type of research: 1) small sample sizes, incomplete data, and inconsistencies in the data, 2) subjective nature of some risk factors, and 3) measurement scales that for some risk factors are at most ordinal.

3. RELATED RESEARCH

A number of useful related reliability and maintenance measurement projects have been reported in the literature. Much of the research and literature in software metrics concerns the measurement of code characteristics [10, 12]. This is satisfactory for evaluating product quality and process effectiveness once the code is written. However, if organizations use measurement plans that are limited to measuring code, these plans will be deficient in the following ways: incomplete, lack coverage (e.g., no requirements analysis and design), and start too late in the process. For a measurement plan to be effective, it must start with requirements and continue through to operation and maintenance. Since requirements characteristics directly affect code characteristics and hence reliability and maintainability, it is important to assess their impact when requirements are specified. Briand, et al, developed a process to characterize software maintenance projects [2]. They present a qualitative and inductive methodology for performing objective project characterizations to identify maintenance problems and needs. This methodology aids in determining causal links between maintenance problems and flaws in the maintenance organization and process. Although the authors have related ineffective maintenance practices to organizational and process problems, they have not made a linkage to risk assessment.

Pearse and Oman applied a maintenance metrics index to measure the maintainability of C source code before and after maintenance activities [15]. This technique allowed the project engineers to track the "health" of the code as it was being maintained. Maintainability is assessed but not in terms of risk assessment.

Pigoski and Nelson collected and analyzed metrics on size, trouble reports, change proposals, staffing, and trouble report and change proposal completion times [17]. A major benefit of this project was the use of trends to identify the relationship between the productivity of the maintenance organization and staffing levels. Although productivity was addressed, risk assessment was not considered.

Sneed reengineered a client maintenance process to conform to the ANSI/IEEE Standard 1291, Standard for Software Maintenance [19]. This project is a good example of how a standard can provide a basic framework for a process and can be tailored to the characteristics of the project environment. Although applying a standard is an appropriate element of a good process, risk assessment was not addressed.

Stark collected and analyzed metrics in the categories of customer satisfaction, cost, and schedule with the objective of focusing management's attention on improvement areas and tracking improvements over time [20]. This approach aided management in deciding whether to include changes in the current release, with possible schedule slippage, or include the changes in the next release. However, the author did not relate these metrics to risk assessment.

An indication of the back seat that software risk assessment takes to hardware, Fragola reports on probabilistic risk management for the Space Shuttle. Interestingly, he says: "The shuttle risk is embodied in the performance of its hardware, the careful preparation activities that its ground support staff take between flights to ensure this performance during a flight, and the procedural and management constraints in place to control their activities." [4]. There is not a word in this statement or in his article about software! Another hardware-only risk assessment is by Maggio, who says: "The current effort is the first integrated quantitative assessment of the risk of the loss of the shuttle vehicle from 3 seconds prior to liftoff to wheel-stop at mission end." Again, not a word about software [9]. Pfleeger lays out a roadmap for assessing project risk that includes risk prioritization [16], a step that we address with the degree of confidence in the statistical analysis of risk (see Section 6). This paper is organized as follows: research approach, risk factors, results, and conclusions.

4. RESEARCH APPROACH

By retrospectively analyzing the relationship between requirements and reliability, we were able to identify those risk factors that are associated with reliability and we were able to prioritize them based on the degree to which the relationship was statistically significant. In order to quantify the effect of a requirements change, we use various risk factors that are defined as the attribute of a requirement change that can induce adverse effects on reliability (e.g., failure incidence), maintainability (e.g., size and complexity of the code), and project management (e.g. personnel resources). Various examples of risk factors are shown in Section 5.

Table 1 shows the Change Request Hierarchy of the Space Shuttle, involving change requests (i.e., a requests for a new requirement or modification of an existing requirement), discrepancy reports (i.e., reports that document deviations between specified and observed software behavior), and failures. We analyzed categories 1 versus 2 with respect to risk factors as discriminants of the categories.

Table 1: Change Request Hierarchy

Change Requests (CRs)

- 1. No Discrepancy Reports (i.e., CRs with no DRs)
- 2. (Discrepancy Reports) or (Discrepancy Reports and Failures)
  - 2.1 No failures (i.e., CRs with DRs only)
  - 2.2 Failures (i.e., CRs with DRs and Failures)
    - 2.2.1 Pre-release failures
    - 2.2.2 Post-release failures

4.1 Categorical Data Analysis

Using the null hypothesis, Ho: A risk factor is not a discriminator of reliability versus the alternate hypothesis H1: A risk factor is a discriminator of reliability, we used categorical data analysis to test the hypothesis. A similar hypothesis was used to assess whether risk factors can serve as discriminators of metrics characteristics. We used the requirements, requirements risk factors, reliability, and metrics data we have from the Space Shuttle "*Three Engine Out*" software (abort sequence invoked when three engines are lost) to test our hypotheses. Samples of these data are shown below.

- Pre-release and post release failure data from the Space Shuttle from 1983 to the present. An example of post-release failure data is shown in Table 2.

Table 2

Failure Found On Operational Increment	Days from Release When Failure Occurred	Discrepancy Report #	Severity	Failure Date	Release Date	Module in Error
Q	75	1	2	05-19-97	03-05-97	10

- Risk factors for the Space Shuttle *Three Engine Out Auto Contingency* software. This software was released to NASA by the developer on 10/18/95. An example of a partial set of risk factor data is shown in Table 3.

Table 3

Change Request Number	SLOC Changed	Complexity Rating of Change	Criticality of Change	Number of Principal Functions Affected	Number of Modifications Of Change Request	Number of Requirements Issues	Number of Inspections Required	Manpower Required to Make Change
A	1933	4	3	27	7	238	12	209.3 MW

- Metrics data for 1400 Space Shuttle modules, each with 26 metrics. An example of a partial set of metric data is shown in Table 4.

Table 4

Module	Operator Count	Operand Count	Statement Count	Path Count	Cycle Count	Discrepancy Report Count	Change Request Count
10	3895	1957	606	998	4	14	16

Table 5 shows the definition of the Change Request samples that were used in the analysis. Sample sizes are small due to the high reliability of the Space Shuttle. However, sample size is one of the parameters accounted for in the statistical tests that produced significant results in certain cases (see Section 6).

Table 5: Definition of Samples

Sample	**Size**
Total CRs	24
CRs with no DRs	14
CRs with (DRs only) or (DRs and Failures)	10
CRs with modules that caused failures	6
CRs can have multiple DRs, failures, and modules that caused failures. CR: Change Request. DR: Discrepancy Report.	

To minimize the effects of a large number of variables that interact in some cases, a statistical categorical data analysis was performed incrementally. We used only one category of risk factor at a time to observe the effect of adding an additional risk factor on the ability to correctly classify change requests that have discrepancy reports (i.e., a report that documents deviations between specified and observed software behavior) or failures and those that do not. The Mann-Whitney test for difference in medians between categories was used because no assumption need be made about statistical distribution; in addition, some risk factors are ordinal scale quantities (e.g., modification level). Furthermore, because some risk factors are ordinal scale quantities, rank correlation was used to check for risk factor dependencies.

5. RISK FACTORS

One of the software process problems of the NASA Space Shuttle Flight Software organization is to evaluate the risk of implementing requirements changes. These changes can affect the reliability and maintainability of the software. To assess the risk of change, the software development contractor uses a number of risk factors, which are described below. The risk factors were identified by agreement between NASA and the development contractor based on assumptions about the risk involved in making changes to the software. This formal process is called a risk assessment. No requirements change is approved by the change control board without an accompanying risk assessment. During risk assessment, the development contractor will attempt to answer such questions as: "Is this change highly complex relative to other software changes that have been made on the Space Shuttle?" If this were the case, a high-risk value would be assigned for the complexity criterion. To date this qualitative risk assessment has proven useful for identifying possible risky requirements changes or, conversely, providing assurance that there are no unacceptable risks in making a change. However, there has been no quantitative evaluation to determine whether, for example, high risk factor software was really less reliable than low risk factor software. In addition, there is no model for predicting the reliability of the software, if the change is implemented. Our research addressed both of these issues.

We had considered using requirements attributes like completeness, consistency, correctness, etc. as risk factors [3]. While these are useful generic concepts, they are difficult to quantify. Although some of the following risk factors also have qualitative values assigned, there are a number of quantitative risk factors, and many of the risk factors deal with the execution behavior of the software (i.e., reliability), which is our research interest.

5.1 Space Shuttle Flight Software Requirements Change Risk Factors

The following are the definitions of the nineteen risk factors, where we have placed the risk factors into categories and have provided our interpretation of the question the risk factor is designed to answer. If the answer to a yes/no question is "yes", it means this is a high-risk change with respect to the given risk factor. If the answer to a question that requires an estimate is an anomalous value, it means this is a high-risk change with respect to the given risk factor.

For each risk factor, it is indicated whether there is a statistically significant relationship between it and reliability for the software version analyzed. The details of the findings are shown in Section 6. In many instances, there was insufficient data to do the analysis. These cases are indicated below. The names of the risk factors used in the analysis are given in quotation marks.

Complexity Factors
o Qualitative assessment of complexity of change (e.g., very complex); "complexity". **Not significant.**
- Is this change highly complex relative to other software changes that have been made on the Space Shuttle?

o Number of modifications or iterations on the proposed change; "mods". **Significant.**
- How many times must the change be modified or presented to the Change Control Board (CCB) before it is approved?

Size Factors
o Number of source lines of code affected by the change; "sloc". **Significant.**
- How many source lines of code must be changed to implement the change request?

o Number of modules changed; "mod chg". **Not significant.**
- Is the number of changes to modules excessive?

Criticality of Change Factors
o Criticality of function added or changed by the change request; "crit func" (insufficient data)
- Is the added or changed functionality critical to mission success?

o Whether the software change is on a nominal or off-nominal program path (i.e., exception condition); "off nom path". (insufficient data)
- Will a change to an off-nominal program path affect the reliability of the software?

Locality of Change Factors
o The area of the program affected (i.e., critical area such as code for a mission abort sequence); "critic area" (insufficient data)
- Will the change affect an area of the code that is critical to mission success?

o Recent changes to the code in the area affected by the requirements change; "recent chgs" (insufficient data)
- Will successive changes to the code in one area lead to non-maintainable code?

o New or existing code that is affected; "new\exist code" (insufficient data)
- Will a change to new code (i.e., a change on top of a change) lead to non-maintainable code?

o Number of system or hardware failures that would have to occur before the code that implements the requirement would be executed; "fails ex code" (insufficient data)
- Will the change be on a path where only a small number of system or hardware failures would have to occur before the changed code is executed ?

Requirements Issues and Functions Factors
o Number and types of other requirements affected by the given requirement change (requirements issues); "other chgs" (insufficient data)
- Are there other requirements that are going to be affected by this change? If so, these requirements will have to be resolved before implementing the given requirement.

o Number of possible conflicts among requirements (requirements issues); "issues" **Significant.**

- Will this change conflict with other requirements changes (e.g., lead to conflicting operational scenarios)

o Number of principal software functions affected by the change; "prin funcs" **Not significant.**
- How many major software functions will have to be changed to make the given change?

Performance Factors
o Amount of memory space required to implement the change; "space" **Significant.**
- Will the change use memory to the extent that other functions will not have sufficient memory to operate effectively?

o Effect on CPU performance; "cpu" (insufficient data)
- Will the change use CPU cycles to the extent that other functions will not have sufficient CPU capacity to operate effectively?

Personnel Resources Factors
o Number of inspections required to approve the change; "inspects" **Not significant.**
- Will the number of requirements inspections lead to excessive use of personnel resources?

o Manpower required to implement the change; "manpower" **Not significant.**
- Will the manpower required to implement the software change be significant?

o Manpower required to verify and validate the correctness of the change; "cost" **Not significant.**
- Will the manpower required to verify and validate the software change be significant?

o Number of tests required to verify and validate the correctness of the change; "tests" **Not significant.**
- Will the number of tests required to verify and validate the software change be significant?

6. RESULTS

This section contains the results of performing the following statistical analyses shown in Tables 6, 7, and 8, respectively. Only those risk factors where there was sufficient data and the results were statistically significant are shown.

a. Categorical data analysis on the relationship between CRs with no DRs vs. ((DRs only) or (DRs and Failures)), using the Mann-Whitney Test.
b. Dependency check on risk factors, using rank correlation coefficients; and
c. Identification of modules that caused failures as a result of the CR, and their metric values.

6.1. Categorical Data Analysis

Of the original nineteen risk factors, only four survived as being statistically significant (alpha ≤ .05); seven were not significant; and eight had insufficient data to make the analysis. As Table 6 shows, there are statistically significant results for CRs with no DRs vs. ((DRs only) or (DRs and Failures)) for the risk factors "mods", "sloc", "issues", and "space". We use the value of alpha in Table 6 as a means to prioritize the use of risk factors, with low values meaning high priority. The priority order is: "issues", "space", "mods", and "sloc".

The significant risk factors would be used to predict reliability problems for this set of data and this version of the software. Whether these results would hold for future versions of the software would be determined in validation tests in future research. The finding regarding "mods" does confirm the software developer's view that this is an important risk factor. This is the case because if there are many iterations of the change request, it implies that it is complex and difficult to understand. Therefore, the change is likely to lead to reliability problems. It is not surprising that the size of the change "sloc" is significant because our previous studies of Space Shuttle metrics have shown it to be important [18]. Conflicting requirements "issues" could result in reliability problems when the change is implemented. The on-board computer memory required to implement the change "space" is critical to reliability because unlike commercial systems, the Space Shuttle does not have the luxury of large physical memory, virtual memory, and disk memory to hold its programs and data. Any increased requirement on its small memory to implement a change comes at the price of demands from competing functions.

Table 6: Statistically Significant Results (alpha ≤ .05). CRs with no DRs vs. ((DRs only) or (DRs and Failures)). Mann-Whitney Test

Risk Factor	**Alpha**	**Median Value** CRs with no DRs	**Median Value** (DRs only) or (DRs and Failures)
issues	.0076	1	14
space	.0186	6	123
mods	.0401	0	4
sloc	.0465	10	88.5

issues: Number of possible conflicts among requirements.
space: Amount of memory space required to implement the change.
mods: Number of modifications of the proposed change.
sloc: Number of source lines of code affected by the change.

In addition to identifying predictive risk factors, we must also identify thresholds for predicting when the number of failures would become excessive (i.e., rise rapidly with the risk factor). An example is shown in Figure 1 where cumulative failures are plotted against cumulative issues. The figure shows that when issues reach 286, failures reach 3 (obtained by querying the data point) and climb rapidly thereafter. Thus, an issues count of 286 would be the best estimate of the threshold to use in controlling the quality of the next version of the software. This process would be repeated across versions with the threshold being updated as more data is gathered. Thresholds would be identified for each risk factor in Table 6. This would provide multiple alerts for the quality of the software going bad (i.e., the reliability of the software would degrade as the number of alerts increases).

6.2. Dependency Check on Risk Factors

In order to check for possible dependencies among risk factors that could confound the results, rank correlation coefficients were computed in Table 7. Using an arbitrary threshold of .7, the results indicate a significant dependency among "issues", "mods", and "sloc" for CRs with no DRs. That is, as the number of conflicting requirements increases, the number of modifications and size of the change increase. In addition, there is a significant dependency among "space", "mods", and "issues" for (DRs only) or (DRs and Failures). That is, as the number of conflicting requirements increases, the memory space and the number of modifications increase.

Table 7: Rank Correlation Coefficients of Risk Factors

	CRs with no DRs			
	mods	sloc	issues	space
mods		.370	**.837**	.219
sloc	.370		**.717**	.210
issues	**.837**	**.717**		.026
space	.219	.210	.026	
(DRs only) or (DRs and Failures)				
	mods	sloc	issues	space
mods		.446	.363	**.759**
sloc	.446		.602	.569
issues	.363	.602		**.931**
space	**.759**	.569	**.931**	

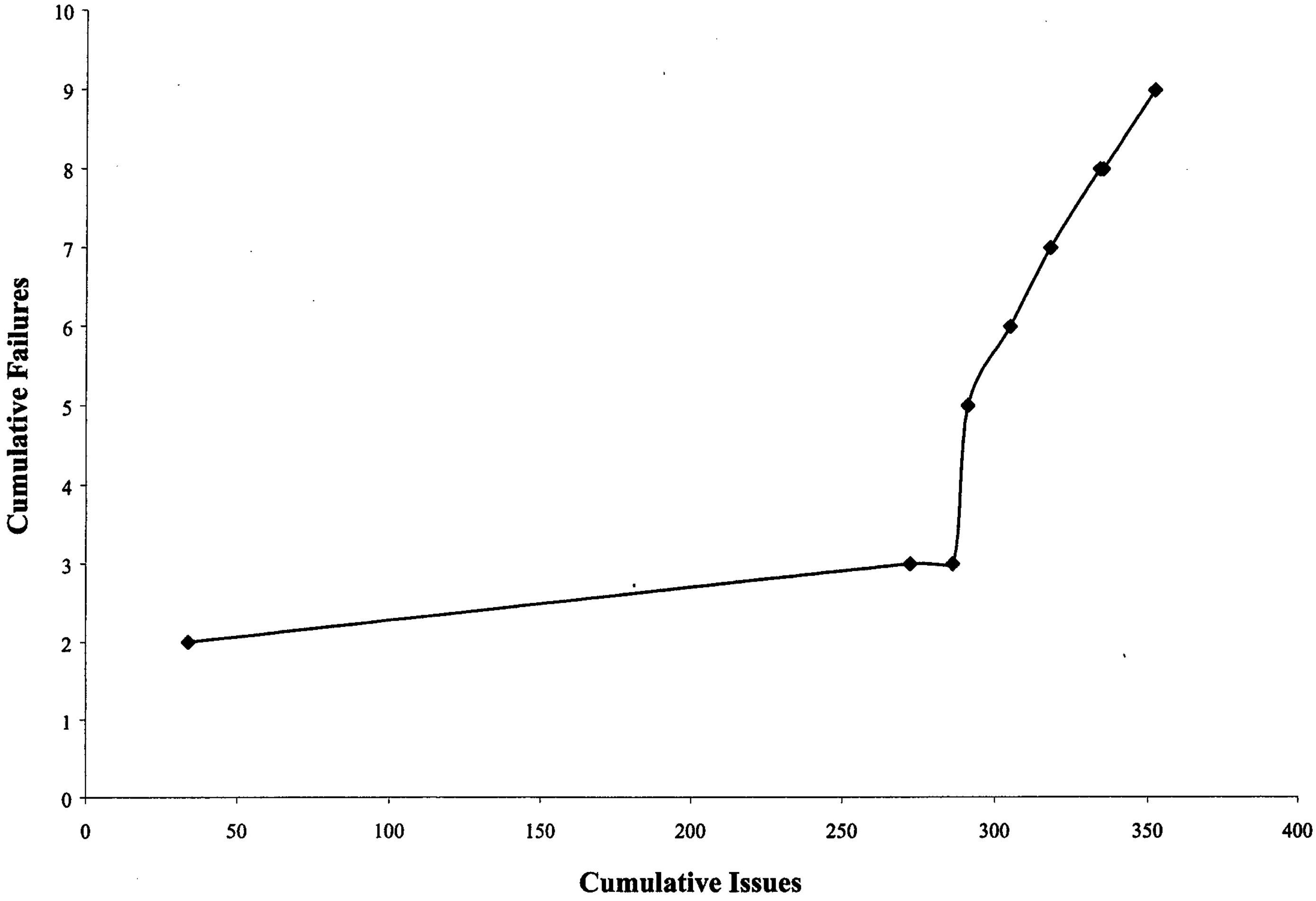


Figure 1. Failures vs. Issues

6.3 Identification of Modules that Caused Failures

Table 8 shows modules that caused failures, as the result of the CRs, had metric values that far exceed the critical values. The latter were computed in previous research [18]. A critical value is a discriminant that distinguishes high quality from low quality software. A module with metric values exceeding the critical values is predicted to cause failures. Although the sample sizes are small, due to the high reliability of the Space Shuttle, the results consistently show that modules with excessive size and complexity lead to failures. Not only will the reliability be low but this software will also be difficult to maintain. The application of this information is that there is a high degree of risk when changes are made to software that has the metric characteristics shown in the table. Thus, these characteristics should be considered when making the risk analysis.

Table 8: Selected Risk Factor Module Characteristics

Change Request	Module	Metric	Metric Critical Value	Metric Value
A	1	change history line count in module listing	63	558
A	2	non-commented loc count	29	408
B	3	executable statement count	27	419
C	4	unique operand count	45	83
D	5	unique operator count	9	33
E	6	node count (in control graph)	17	66
All of the above metrics exceeded the critical values for all of the above Change Requests.				

7. CONCLUSIONS

Risk factors that are statistically significant can be used to make decisions about the risk of making changes. These changes affect the reliability of the software. Risk factors that are not statistically significant should not be used; they do not provide useful information for decision-making and cost money and time to collect and process. The number of requirements issues ("issues"), the amount of memory space required to implement the change ("space"), the number of modifications ("mods"), and the size of the change ("sloc"), were found to be significant, in that priority order. In view of the dependencies among these risk factors, "issues" would be the choice if the using organization could only afford a single risk factor. We also showed how risk factor thresholds are determined for controlling the quality of the next version of the software.

Statistically significant results were found for CRs with no DRs vs. ((DRs only) or (DRs and Failures)).

Metric characteristics of modules should be considered when making the risk analysis because metric values that exceed the critical values are likely to result in unreliable and non-maintainable software.

Our methodology can be generalized to other risk assessment domains, but the specific risk factors, their numerical values, and statistical results may vary. Future research will involve applying the methodology to the next version of the Space Shuttle software and identifying the statistically significant risk factors and thresholds to see whether they match the ones identified in this research.

8. ACKNOWLEDGEMENTS

We acknowledge the technical support received from Julie Barnard, Boyce Reeves, and Patti Thornton of United Space Alliance. We also acknowledge the funding support received from Dr. Allen Nikora of the Jet Propulsion Laboratory.

9. REFERENCES

[1] Barry W. Boehm, "Software Risk Management: Principles and Practices", IEEE Software, Vol. 8, No. 1, January 1991, pp. 32-41.

[2] Lionel C. Briand, Victor R. Basili, and Yong-Mi Kim, "Change Analysis Process to Characterize Software Maintenance Projects", Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 38-49.

[3] Alan Davis, Software Requirements: Analysis and Specifications, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[4] Joseph R. Fragola, "Space Shuttle Program Risk Management", Proceedings Annual Reliability and Maintainability Symposium, 1996, pp. 133-142.

[5] Taghi M. Khoshgoftaar and Edward B. Allen, "Predicting the Order of Fault-Fault-Prone Modules in Legacy Software", Proceedings of the Ninth International Symposium on Software Reliability Engineering, November 4-7, 1998, Paderborn, Germany, pp. 344-353.

[6] Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, and Gary P. Trio, "Detection of Fault-Prone Software Modules During a Spiral Life Cycle", Proceedings of the International Conference on Software Maintenance, November 4-8, 1996, Monterey, California, pp. 69-76.

[7] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai Kalaichelvan, and Nishith Goel, "Early Quality Prediction: A Case Study in Telecommunications", IEEE Software, Vol. 13, No. 1, January 1996, pp. 65-71.

[8] D. Lanning and T. Khoshgoftaar, "The Impact of Software Enhancement on Software Reliability", IEEE Transactions on Reliability, Vol. 44, No. 4, December 1995, pp. 677-682.

[9] Gaspare Maggio, "Space Shuttle Probabilistic Risk Assessment Methodology and Application", Proceedings Annual Reliability and Maintainability Symposium, 1996, pp. 121-132.

[10] Sebastian G. Elbaum and John C. Munson, "Getting a Handle on the Fault Injection Process: Validation of Measurement Tools", Proceedings of the Fifth International Software Metrics Symposium, November 20-21, 1998, Bethesda, Maryland, pp. 133-141.

[11] John C. Munson and Darrell S. Werries, "Measuring Software Evolution", Proceedings of the Third International Software Metrics Symposium, March 25-26, 1996, Berlin, Germany, pp. 41-51.

[12] Allen P. Nikora, Norman F. Schneidewind, and John C. Munson, IV&V Issues in Achieving High Reliability and Safety in Critical Control Software, Final Report, Volume 1 – Measuring and Evaluating the Software Maintenance Process and Metrics-Based Software Quality Control, Volume 2 – Measuring Defect Insertion Rates and Risk of Exposure to Residual Defects in Evolving Software Systems, and Volume 3 – Appendices, Jet Propulsion Laboratory, National Aeronautics and Space Administration, Pasadena, California, January 19, 1998.

[13] Magnus C. Ohlsson and Claes Wohlin, "Identification of Green, Yellow, and Red Legacy Components", Proceedings of the International Conference on Software Maintenance, November 16-20, 1998, Bethesda, Maryland, pp. 6-15.

[14] Niclas Ohlsson and Hans Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches", IEEE Transactions on Software Engineering, Vol. 22, No. 12, December 1996, pp. 886-894.

[15]Troy Pearse and Paul Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", Proceedings of the International Conference on Software Maintenance, Opio (Nice), France, October 17-20, 1995, pp. 295-303.

[16] Shari Lawrence Pfleeger, "Assessing Project Risk", Software Tech News, Dod Data Analysis Center for Software, vol.2, no. 2, pp. 5-8.

[17]Thomas M. Pigoski and Lauren E. Nelson, "Software Maintenance Metrics: A Case Study", Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, September 19-23, 1994, pp. 392-401.

[18] Norman F. Schneidewind, "Software quality control and prediction model for maintenance", Annals of Software Engineering, Baltzer Science Publishers, Volume 9 (2000), May 2000, pp. 79-101.

[19] Harry Sneed, "Modelling the Maintenance Process at Zurich Life Insurance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 217-226.

[20]George E. Stark, "Measurements for Managing Software Maintenance", Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4-8, 1996, pp. 152-161.

[21] Webster's New Universal Unabridged Dictionary, Second Edition, Simon and Shuster, New York, 1979.

Design for Independent Composition and Evaluation of High-Confidence Embedded Software Systems

Farokh B. Bastani
I-Ling Yen
University of Texas at Dallas
bastani@utdallas.edu
ilyen@utdallas.edu
(972)883-{2299,6446}

John Linn
Texas Instruments
linn@ti.com
(214)480-6262

Kashi Rao
Alcatel USA
Kashi.Rao@usa.alcatel.com
(972)996-7375

Victor L. Winter
Sandia National Labs.
vlwint@sandia.gov
(505)284-2696

Abstract

Embedded computer systems are systems in which one or more computers monitor and control a larger electromechanical system. Applications include telecommunication systems, health care systems, defense systems, manufacturing automation systems, etc. Many of these systems have mission-critical requirements and, hence, it is necessary to have high confidence in their reliability before deploying them. Further, due to variable environmental conditions and rapid technological advances, it is necessary to design these systems to be adaptable and easily modifiable.

One way of achieving these objectives is to decompose a complex system into smaller subsystems. Several decomposition methods have been developed. However, most of these methods do not necessarily enable system properties to be inferred from subsystem properties. In this paper, we constrain each subsystem to be an Independently Developable and End-user Assessable Logical (IDEAL) subsystem. We classify a system of IDEAL subsystems into three classes depending on the interaction pattern among the subsystems. Then, we present an extended finite state machine notation for modeling these subsystems and show how they can be statically composed together to form the system. The paper also shows how system-level properties can be computed from IDEAL subsystem-level properties.

Keywords: Embedded software systems, High-assurance methods, Software decomposition and composition, Software reliability assessment.

1 INTRODUCTION

In recent years, dramatic advances in technology have made it possible to envision and develop highly critical embedded systems, including high-consequence real-time distributed applications such as on-board weapons control systems, avionics and vehicle control systems, air-traffic control systems, manufacturing systems, etc. These embedded systems are becoming increasingly sophisticated and complex. For example, consider telecommunications systems. Just a few years ago, all that a switching system had to do was to establish a route for a call, monitor the call for billing purposes, and release the resources dedicated to the call after it was completed. In recent years, this simple scenario has become extremely complex with an explosive growth in the number of features and capabilities. Telecommunications systems must now handle stationary and mobile calls (both cellular and satellite wireless systems), handle various failure modes (switches, trunk-lines, satellites), support voice and data transmissions, handle different service plans, and provide numerous user-oriented features (call forwarding, speed dialing, caller id, 911 service, etc.). This tendency to continually push the envelope of complexity has been accelerating in many application areas, including defense systems, manufacturing systems, air-traffic control systems, vehicle control systems, etc.

The growing sophistication and complexity of these embedded applications is stretching the limits of current software technology. Almost all the domain-specific knowledge for a given application is now embodied within the software and this is extending down to even traditionally all-hardware systems, such as software radios, digital cameras, and networked appliances. The reason is that software enables the implementation of high-quality, intelligent, flexible, and field upgradable application-specific features over generic hardware components. Software also enhances the robustness of a distributed application by monitoring the environment and adapting the system to tolerate hardware failures, network congestion, security attacks, etc.

Several challenges must be overcome for the development and deployment of practical embedded systems. First, for critical applications, such as mobile command and control systems and vehicle control systems, it is necessary to be able not only to *achieve* high quality but also to rigorously *demonstrate* that high quality has in fact been achieved. Further, due to the rapidly changing technology and intense global competition, industry must be able to develop and field these systems quickly and at a low cost. One method of handling these problems is to decompose the system in to more manageable portions. Various decomposition methods have been developed [7, 11, 17, 21]. All these methods simplify the analysis of software requirements. Several of these methods also facilitate the assurance of software quality, i.e., they result in the identification of subsystems that can be designed and implemented independently of the other subsystems. However, these methods do not necessarily enable the demonstration of high quality For complex software systems, one way of achieving "assessability" is to be able to infer the properties of the system from those of its subsystems. (The subsystems are smaller relative to the entire system and, therefore, easier to evaluate.) However, this inference is not always possible for arbitrary decompositions where, after the implementation phase, we may still be left with the task of determining the reliability of one complex monolithic program.

This paper presents an approach in which software specifications are decomposed into subsubsystems in such a manner that reasoning about the properties possessed by the composition of subsystems can be deduced simply from the properties of the individual subsystems. Each subsystem in this class is *independently developable*, i.e., it can be designed and implemented independently of the other subsystems in the system. In addition, each subsystem is *end-user assessable*, i.e., it can be tested or verified by the end-user independently of any other subsystem. That is, the end-user (not the developers!) can certify each subsystem by directly observing and comparing its behavior with the expected behavior even though it might be part of a large system. We refer to these subsystems as IDEAL (Independently Developable End-user Assessable Logical) software subsystems.

The rest of this paper is organized as follows. Section 2 presents the possible interaction patterns between IDEAL subsystems. Section 3 presents static and dynamic methods of composing a system of IDEAL subsystems. Section 4 presents assurance methods and Section 5 discusses some related work. Finally, Section 6 summarizes the paper and lists some possible research areas.

2 SYSTEM MODEL

We consider a system that consists of a collection of autonomous processes, $\{P_0, P_1, \cdots, P_n\}$. The following two properties must be satisfied to ensure that system-level properties can be inferred from subsystem-level properties.

1. *End-user observability.* It must be possible to directly observe and evaluate the behavior of each process irrespective of whether it is running in isolation or in conjunction with other processes. In our approach, each P_i reads from or writes to a shared state space. The shared state space, such as the physical environment for process-control systems, is visible to the end-user.

2. *Implementation-invariant state space.* The specification of each P_i must be such that the distribution of the state space is statistically invariant to any *correct* implementation of P_i. Essentially, this means that it should be possible to guarantee the absence of nondeterminism in the behavior of every P_i.

Let $A = \{a_1, a_2, \cdots, a_k\}$ denote the set of actuators and $S = \{s_1, s_2, \cdots, s_l\}$ denote the set of sensors in the system. An actuator a affects a subset of the state space; let this be denoted by c_a, i.e., the "capability" set for actuator a. At time t, an IDEAL process, P_i, monitors a subset $S_{P_i}(t)$ of sensors and sends commands to a subset $A_{P_i}(t)$ of actuators. The instantaneous capability set for process P_i is a time-varying subset of the state space, $C_{P_i}(t)$, such that $C_{P_i}(t) = \cup_{a \in A_{P_i}(t)} c_a$. The complete capability set for process P_i is the entire subset of the state space that it can ever impact; this is given by $C_{P_i} = \cup_{a \in A_{P_i}} c_a$ where $A_{P_i} = \{a | \exists t : t \geq 0 :: a \in A_{P_i}(t)\}$. A system of cooperating IDEAL processes consists of a set of IDEAL processes, $\mathbf{P} = \{P_0, P_1, \cdots, P_n\}$.

P_i and P_j have producer/producer interaction if they affect at least one common point in the state space. They have producer/consumer interaction if one process generates information (or causes state changes) that is used as input by the other process. These types of processes are very useful for cooperative situations where one process performs a task that is then completed by another process.

The composition of two processes, P_i and P_j depends on their capability sets. We have classified these into three distinct cases, namely, Spatially Separable (SS), Spatially Inseparable Temporally Separable (SITS), and Spatially Inseparable Temporally Inseparable (SITI) IDEAL processes.

Spatially Separable (SS) Processes. Two processes P_i and P_j are *spatially separable* if $C_{P_i} \cap C_{P_j} = \phi$, that is, if they always affect different regions of the state space. Spatially separable processes will *never* interfere with each other, so no special composition is required. They can simply work and coexist with each other. This case is not theoretically interesting, but it is very useful and effective in practice.

Spatially Inseparable Temporally Inseparable (SITI) Processes. In this case, two processes need to simultaneously cause changes in the same portion of the state space. For example, one process may be concerned with the safety requirements of the system while another process may be concerned with the functional requirements of the system. They may both be trying to influence the control trajectory but, perhaps, toward different goals. In these cases, only producer/producer interaction is meaningful.

Consider a control program P. Let $\mathbf{S}(t)$ denote the time-varying state space of the system and $s(t)$ denote the actual state of the system at time t. $s(t)$ describes a **trajectory** of the system through its state space. The state space can be divided into **goal** states (states that the control system should reach), **constraint** states (states that the system must avoid), and **free** states (all the other states). The purpose of the control program is to determine a trajectory that passes through the free states and reaches a goal state without passing through any constraint states.

Our experiences suggest that the top level requirements specification, g, for control systems can be decomposed into a **conjunction** of predicates, $g = g_1 \wedge g_2 \wedge \cdots \wedge g_n$. The individual predicates, g_i, can be further decomposed into a **disjunction** of predicates, i.e., $g_i = g_{i1} \vee g_{i2} \vee \cdots \vee g_{in_i}$. Note that conjunctive and disjunctive decompositions can be applied to the specification iteratively as necessary.

Let $\mathbf{S_{ij}}(t)$ denote the view of the state space that only reflects the goal or constraint specified by predicate g_{ij}. That is, $\mathbf{S_{ij}}(t)$ has the same state space as $\mathbf{S}(t)$, but all its states other than those in $\{x | x \in \mathbf{S}(t) \wedge g_{ij}(x)\}$ are free states. Let P_{ij} be a program (corresponding to an IDEAL process) that solves the limited control problem expressed in $\mathbf{S_{ij}}(t)$. Given a specification of the form described above, an important question is: "How to compose the P_{ij}'s, $1 \leq i \leq n$, $1 \leq j \leq n_i$, and how to validate and reason about the properties of the system resulting from the composition?" In conventional programs, P_{ij} is viewed as a *function* that maps its input domain to its output space, i.e., P_{ij} returns a single value for a given execution. There is no obvious mathematical model for merging independently developed P_{ij}'s into the overall system P since the output of the P_{ij}'s may be incompatible with each other. For example, the code for a safety process for a vehicle-control system may simply set the system to a fixed state that is guaranteed to be safe. This ensures that the goal of the safety process will be achieved since the system is always in a safe state. However, it prevents anything useful from being done.

To cope with this problem, we view each P_{ij} as a general relation, i.e., it returns the set of all possible output values for each input. We refer to P_{ij} as a relational program [3]. Viewing programs as general relations greatly simplifies the composition procedure and reasoning about the system behavior. Now, P can be obtained by simply forming the intersection of the output sets of P_i's, or $P \equiv P_1 \cap P_2 \cap \cdots \cap P_n$, where P_i is the program for achieving g_i. Similarly, each P_i can be obtained via a systematic *union* operation over its subsystems, i.e., $P_i \equiv P_{i_1} \cup P_{i_2} \cup \cdots \cup P_{in_i}$.

Among various subgoals of a control system, it is frequently the case that one goal has a higher priority than the other. For example, safety is a goal that has to be assured to a high degree of confidence in many control applications. If the output of the safety control process conflicts with the output of another process, then the safety goal should prevail. To ensure satisfaction of critical properties in the system, we introduce a priority scheme into relational composition. Consider a software architecture consisting of a collection of IDEAL processes, P_0, P_1, $\cdots$, P_n. Let $O_j(x)$ denote the output set of $P_j(x)$, $0 \leq j \leq n$, for input x. Assume that the processes are prioritized according to their criticality and $priority(P_i) > priority(P_j)$ for all $i < j$. We define $Q_j(x)$ to be the intersection of the output sets for P_0 through P_j, i.e., $Q_j(x) = O_0(x) \cap O_1(x) \cap \cdots \cap O_j(x)$. Clearly, if ϕ denotes the empty set, then $Q_j(x) = \phi \Rightarrow Q_k(x) = \phi$, for all $k > j$. Let *choose* denote a deterministic function that takes a nonempty set and selects an element from the set, i.e., for set $S \neq \phi$, $choose(S) \in S$. To achieve prioritization, the overall output of the system can be defined as $choose(Q_k(x))$ where $k = max\{j | 0 \leq j \leq n \wedge Q_j(x) \neq \phi\}$. In this design, the system output will never violate P_0, so P_0 should be designed to implement the critical functions of the system. Each P_i, for $i > 0$, can then be designed to implement some functional requirements of the system or to provide a better quality of service, such as control trajectories that conserve resources, achieve more refined control, etc.

Spatially Inseparable Temporally Separable (SITS) Processes. In a SITS system, there is at least one point in the state space that is affected by both P_i and P_j. Hence, $C_{P_i} \cap C_{P_j} \neq \phi$. However, P_i and P_j can be separated in time so that they do not simultaneously access the same point. Formally, $\forall t : t \geq 0 :: C_{P_i}(t) \cap C_{P_j}(t) = \phi$. One way of achieving this separation is to create a third process, P_k, such that P_k coordinates P_i and P_j to ensure that they do not interfere with each other. We refer to IDEAL processes of type P_k as coordination processes. The simplest coordination processes are those that only ensure mutual exclusion. More complex processes also ensure the absence of deadlocks.

The case considered in the previous section is conceptually simple. In every cycle, each process looks at the state space (via sensors), performs its computation, and updates the state space (via actuators). Thus, there is no functional dependency between the processes, i.e., one process does not depend on the output of another process.

Similarly, since all the processes execute synchronously, there is no synchronization dependency among the processes.

The class of Spatially Inseparable Temporally Separable systems is a richer class than SITI systems. Since actions can be separated in time, it is now possible to have functional dependency among the processes, i.e., a producer process can generates data that is subsequently used by a consumer process. Likewise, since any two SITS processes, P_i and P_j, may interfere with each other when they access the shared state space, there is a need for some sort of coordination to ensure that their accesses are serializable. Usually, this type of coordination logic is interspersed within the code that implements the functional aspects of the process. Two exceptions are the behavioral classes in DRAGOON [2] and the "separation of concerns" approach in Aspect-Oriented Programming [16]. There are significant advantages in separating out the coordination details from the implementation of functional aspects. First, it significantly reduces the state space of each process which makes reliability assessment much simpler — processes corresponding to the functional aspects of the system only need to be concerned with the application state space while coordination processes are only concerned with aspects related to process synchronization. Second, it enhances the reusability and maintainability of the system. In particular, the coordination subsystems, which normally require more expertise to write, can be reused in a variety of different contexts. It also makes the system extremely flexible — the coordinators can be changed without having to recode any other portion of the system, either the functional processes or the access methods for the shared object. Third, with some enhancements, it makes all functional as well as coordination processes IDEAL processes.

3 CODE SYNTHESIS

It is difficult to automatically convert goal-oriented specifications into code without a substantial knowledge-base (and associated inference engine) for the application domain. There are various formalisms that one can use to express system knowledge. One practical approach is to use a Finite State Machine paradigm to describe the state space of a system.

Definition 1. A **Finite State Machine (FSM)**model for a process-control system consists of a set of states, $S = \{s_1, s_2, \cdots, s_k\}$ and a set of transitions, $T = \{t_{ij} | 1 \leq i, j \leq k\}$. Associated with each transition, t_{ij}, is a control command vector, $guard(t_{ij})$. The meaning of t_{ij} is that if the state of the machine is s_i and $guard(s_{ij})$ has the given value, then the new state will be s_j [20].

Given the presence of continuous variables in many control systems, it is necessary to extend the FSM model to a Hybrid FSM (HFSM) model where continuous variables are associated with each state.

Definition 2. A **Hybrid FSM (HFSM)** is an FSM where a vector of continuous variables is associated with each state. A transition t_{ij} is an **instantaneous** transition if the state changes to s_j whenever $guard(t_{ij})$ has the given value. A transition t_{ij} is an **eventually** transition if the state will eventually change to s_j and $guard(t_{ij})$ has the given value continuously in state s_i. A state is a **boundary** state if all transitions are instantaneous; otherwise, it is an **interior** state.

We call an HFSM knowledge base of a system a*model* of that system. A model when combined with a specification provides the basis for formal software development. The subgoals that form the literals of our behavioral decomposition can be specified against the model in a non-algorithmic manner using traditional techniques (e.g., pre/postconditions, invariants, constraints, etc.).

Given a non-algorithmic specification, techniques such as deductive synthesis, refinement based transformation, etc., can be used to create an abstract algorithm satisfying the specification. The resulting abstract algorithm is essentially a path through the HFSM model of the system and, as such, it can be executed to provide early validation.

In order to fit the notion of an abstract algorithm within our relation-based model of computation, we further extend the HSFM and introduce relational HSFMs defined as follows.

Definition 3. A **Relational HFSM (RHFSM)** is an HFSM where $guard(t_{ij})$ can be a set of values rather than a single value.

Given an RHFSM, it is possible to automatically generate the code through transformation.

3.1 A Simple Example

To illustrate the RHFSM model and the power of IDEAL subsystems, consider the following simplified example of a vehicle control system (see Figure 1). At time t, the vehicle is at location $x(t)$ and has velocity $v(t)$. The road intersects with a railway track from position 0 to position d. Input $alarm(t)$ is $true$ if a train is in the crossing zone. Input $mrt(t)$ gives the most recent time that a train entered the crossing zone. It is used to calculate the maximum available reaction time, $\delta(t)$, for the vehicle; specifically, at time t, $\delta(t) = \delta_0$ if $alarm(t)$ is $false$ and

$\delta(t) = max\{0, \delta_0 - (t - mrt(t))\}$ if $alarm(t)$ is *true*. The goal of the controller is to drive the vehicle to position y without any accidents and subject to other constraints, such as obeying speed limit laws and providing a smooth ride. There is only one actuator; it accepts values for the desired acceleration, $a(t)$, and operates the accelerator or the brake to achieve acceleration $a(t)$.

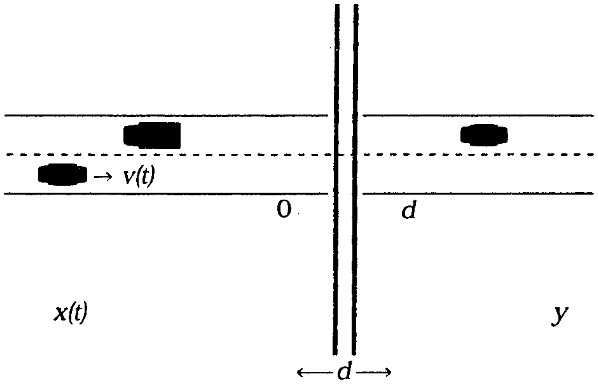


Figure 1: Vehicle control problem.

The requirements of the vehicle control program can be decomposed into four predicates.

1. (g_0) *Safety constraint.* $\forall t : t \geq 0 :: \neg(alarm(t) \wedge 0 < x(t + \delta(t)) < d)$.
2. (g_1) *Reach destination goal.* $\exists t_0 : t_0 \geq 0 :: (\forall t : t \geq t_0 :: (x(t) = y \wedge v(t) = 0))$.
3. (g_2) *Speed limit constraint.* $\forall t : t \geq 0 :: -V_{max} \leq v(t) \leq V_{max}$.
4. (g_3) *Smoothness of ride optimization.* $\forall t : t \geq 0 :: |\frac{d^2v(t)}{dt^2}| \leq a_0$.

We assume that the most critical requirement is g_0 followed by g_1, g_2, and g_3.

Now, we can specify the model for achieving g_i independently using RHSFM. The model for g_2 for the vehicle control program is shown in Figure 2.

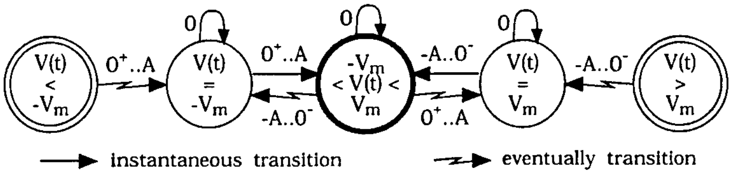


Figure 2: Relational HFSM for g_2.

Here, 0^+ indicates a positive value just larger than 0 and 0^- indicates a negative value just smaller than 0. There are five states, namely, $v(t) < -V_m$, $v(t) = -V_m$, $-V_m < v(t) < V_m$, $v(t) = V_m$, and $v(t) > V_m$, and two control trajectories. One trajectory specifies that in state $v(t) < -V_m$, the acceleration can have any positive value and eventually the system will reach the state $v(t) = -V_m$. In this state, the legal accelerations are $0 \cdot \cdot A$. Similarly, in state $v(t) > V_m$, the acceleration can have any negative value and eventually the system will reach the state $v(t) = V_m$ where the legal accelerations are $-A \cdot \cdot 0$. Any acceleration is legal in the goal state, $-V_m < v(t) < V_m$.

The RHFSM in Figure 2 can be transformed into the following code:

```
P_2: if $v(t) \leq -V_{max} \rightarrow a(t) := 0^+ \cdot\cdot A$[]
        $V_{max} \leq v(t) \rightarrow a(t) := -A \cdot\cdot 0^-$
        $-V_{max} < v(t) < V_{max} \rightarrow a(t) := -A \cdot\cdot A$[]
     end if
```

3.2 Static Composition

We assume we are given an abstract relational algorithm of the form:$g_1 \diamond g_2 \diamond \ldots \diamond g_n$, where each symbol $\diamond$ denotes either a relational conjunction (i.e., intersection) or a relational disjunction (i.e., union), and g_1 through g_n are guarded command lists of the form:

$$\{guard_i \to (c_1, c_2, \cdots, c_j) := (set_{1_i}, set_{2_i}, \cdots, set_{j_i}) | 1 \leq i \leq n\}.$$

Our objective is to apply correctness preserving transformations to an algorithm of the form $g_1 \diamond g_2 \diamond \ldots \diamond g_n$ to obtain a semantically equivalent algorithm consisting of a single list of guarded commands where all $\diamond$ operations have been removed.

Several axioms can be derived and used to simplify the computation of the intersection and union of relational programs. Some of these are listed in the following.

1. If a_0 is a singleton or null then $(a := a_0) \cap Q \equiv (a := a_0)$ for all Q.
2. If U is the universe of the output set, then $(a := U) \cap Q \equiv Q$ for all Q.
3. If $A_1, \cdots, A_n$ are mutually disjoint sets and the actions of Q are of the form $a := \cup_{j=1}^{m} A_{i_j}$ for $\{i_j | 1 \leq j \leq m\} \subset 1 \cdot\cdot n$, then $(a := A_k) \cap Q \equiv (a := A_k)$ for $1 \leq k \leq n$.
4. If the actions of P are all either singletons or ϕ, then $P \cap Q \equiv P$.
5. If $A_1, \cdots, A_n$ are mutually disjoint sets and the actions of P are of the form $a := A_k$, $1 \leq k \leq n$, and those of Q are of the form $a := \cup_{j=1}^{m} A_{i_j}$ for $\{i_j | 1 \leq j \leq m\} \subset 1 \cdot\cdot n$, then $P \cap Q \equiv P$.
6. If U is the universe of the output set, then $(a := U) \cup Q \equiv (a := U)$ for all Q.
7. $(a := \phi) \cup Q \equiv Q$, for all Q, where ϕ denotes the empty set.

Our experience so far has raised some interesting questions as to what kind of set representations (e.g., predicate based, inductive equations) can be handled at the transformational level. This in turn raises interesting questions regarding properties between controlled variables and monitored variables (e.g., continuity).

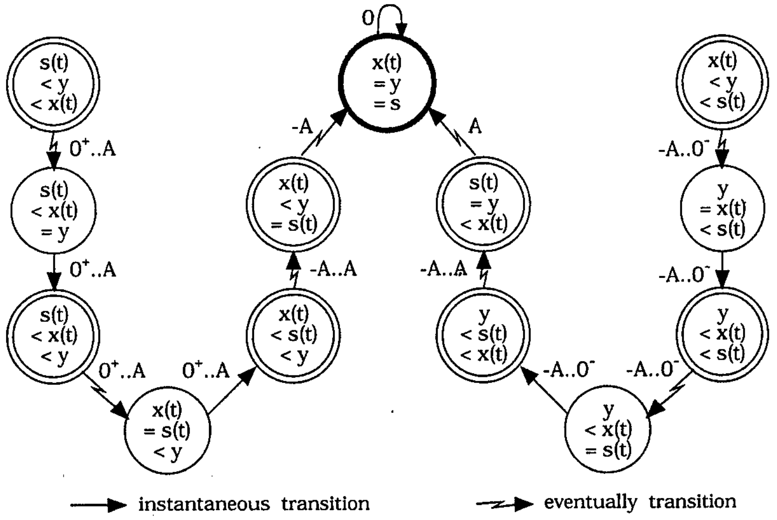


Figure 3: Relational HFSM for g_1.

Examples of Composition of IDEAL Subsystems. Figure 3 shows the relational HFSM for goal g_1 for the vehicle control problem discussed earlier. It consists of two trajectories, corresponding to whether the car is to the left or to the right of the destination. $s(t)$ denotes the stopping position at maximum deceleration. The code, P_1, for g_1 can be automatically derived from this RHFSM and is shown in Figure 4(a).

```
if $x(t) = y = s(t) \rightarrow a(t) := \{0\}[]$
    $x(t) \leq y < s(t) \vee y < x(t) \leq s(t) \rightarrow$
            $a(t) := -A \cdot \cdot 0^-[]$
    $s(t) < y \leq x(t) \vee s(t) \leq x(t) < y \rightarrow$
            $a(t) := 0^+ \cdot \cdot A[]$
    $x(t) < s(t) < y \vee y < s(t) < x(t) \rightarrow$
            $a(t) := -A \cdot \cdot A[]$
    $x(t) < y = s(t) \rightarrow a(t) := \{-A\}[]$
    $s(t) = y < x(t) \rightarrow a(t) := \{+A\}$
end if
```

Figure $4(a)$. Code for P_1.

```
if $x(t) = y = s(t) \rightarrow a(t) := \{0\}[]$
  $x(t) \leq y < s(t) \vee y < x(t) \leq s(t) \rightarrow a(t) := -A \cdot \cdot 0^-[]$
  $s(t) < y \leq x(t) \vee s(t) \leq x(t) < y \rightarrow a(t) := 0^+ \cdot \cdot A[]$
  $x(t) < s(t) < y \vee y < s(t) < x(t) \rightarrow$
      if $v(t) < -V_{max} \rightarrow a(t) := 0^+ \cdot \cdot A[]$
        $-V_{max} \leq v(t) \leq V_{max} \rightarrow a(t) := -A \cdot \cdot A[]$
        $V_{max} < v(t) \rightarrow a(t) := -A \cdot \cdot 0^-$
      end if []
  $x(t) \leq y \leq s(t) \rightarrow a(t) := \{-A\}[]$
  $x(t) \leq y \leq x(t) \rightarrow a(t) := \{+A\}$
end if
```

Figure $4(b)$. Code for $P_1 \cap P_2$.

```
if $x(t) \leq 0 \rightarrow$
      if $s(t) < 0 \rightarrow a(t) := -A \cdot \cdot A[]$
          $s(t) = 0 \rightarrow$
            if $x(t) = s(t) \rightarrow a(t) := -A \cdot \cdot 0[]$
                $x(t) < s(t) \rightarrow a(t) := \{-A\}$
            end if []
          $s(t) > 0 \rightarrow a(t) := \phi$
      end if []
   $0 < x(t) < d \rightarrow a(t) := \phi$ []
   $x(t) \geq d \rightarrow$ ( $\cdots$ similar to $x(t) \leq 0 \cdots$ )
end if
```

Figure $5(a)$. Code for P_{01}.

```
if $x(t) \leq 0 \rightarrow$
      if $p(t) < d \rightarrow a(t) := \phi[]$
          $p(t) \geq d \rightarrow a(t) := \alpha_{min}(t) \cdot \cdot A$
      end if []
   $0 < x(t) < d \rightarrow$
      if $p(t) \leq 0 \rightarrow a(t) := -A \cdot \cdot -\alpha_{min}(t)[]$
          $0 < p(t) < d \rightarrow a(t) := \phi[]$
          $p(t) \geq d \rightarrow a(t) := \alpha_{min}(t) \cdot \cdot A$
      end if []
   $x(t) \geq d \rightarrow$ ( $\cdots$ similar to $x(t) \leq 0 \cdots$ )
end if
```

Figure $5(b)$. Code for P_{02}.

```
if $x(t) \leq 0 \rightarrow$
      if $s(t) < 0 \rightarrow a(t) := -A \cdot \cdot A[]$
          $s(t) = 0 \rightarrow$
             if $x(t) = s(t) \rightarrow$
                   if $p(t) < d \rightarrow a(t) := -A \cdot \cdot 0[]$
                       $p(t) \geq d \rightarrow$
                          $a(t) := -A \cdot \cdot 0 \cup \alpha_{min}(t) \cdot \cdot A$
                   end if []
                $x(t) < s(t) \rightarrow$
                   if $p(t) < d \rightarrow a(t) := \{-A\}[]$
                       $p(t) \geq d \rightarrow$
                          $a(t) := \{-A\} \cup \alpha_{min}(t) \cdot \cdot A$
                   end if []
             end if []
          $s(t) > 0 \rightarrow$
             if $p(t) < d \rightarrow a(t) := \phi[]$
                 $p(t) \geq d \rightarrow a(t) := \alpha_{min}(t) \cdot \cdot A$
             end if
      end if []
   $0 < x(t) < d \rightarrow$
      if $p(t) \leq 0 \rightarrow a(t) := -A \cdot \cdot -\alpha_{min}(t)[]$
          $0 < p(t) < d \rightarrow a(t) := \phi[]$
          $p(t) \geq d \rightarrow a(t) := \alpha_{min}(t) \cdot \cdot A$
      end if []
   $x(t) \geq d \rightarrow$ ($\cdots$ similar to $x(t) \leq 0 \cdots$)
end if
```

Figure $5(c)$. Code for $P_0 \equiv P_{01} \cup P_{02}$.

We can obtain the code for$g_1 \wedge g_2$ by computing $P_1 \cap P_2$. Since g_1 has a higher priority in the intersection, we first analyze the leaf nodes of P_1. There are three terminal actions, namely, "$a(t) := \{0\}$", "$a(t) := \{-A\}$", and "$a(t) := \{+A\}$", and the intersection of these with P_2 gives the same output because g_1 has a higher priority than g_2. That is, $(a(t) := \{0\}) \cap P_2 \equiv a(t) := \{0\}$, and similarly for the other two cases. There are three nonterminal actions, namely, "$a(t) := 0^+ \cdot \cdot A$", "$a(t) := -A \cdot \cdot 0^-$", and "$a(t) := -A \cdot \cdot A$". $(a(t) := 0^+ \cdot \cdot A) \cap R_2 \equiv a(t) := 0^+ \cdot \cdot A$ since the intersection with the first two actions of P_2 is $a(t) := 0^+ \cdot \cdot A$ and the intersection with the third action of P_2 is ϕ (the empty set) which causes the output to be that of P_1 since g_1 has a higher priority than g_2. $(a(t) := -A \cdot \cdot A) \cap P_2 \equiv P_2$ since the intersection of $(a(t) := -A \cdot \cdot A)$ with each action of P_2 returns the corresponding action of P_2. Putting all these together, we obtain the code for $P_1 \cap P_2$ shown in Figure 4(b).

For an example of the union operation, consider the safety requirement,g_0, for the vehicle control program. It can be achieved by not entering the intersection or by entering and crossing it within time $\delta(t)$. Formally, the "do not enter the intersection" subgoal, g_{01}, is given by $(x(t) \leq 0 \Rightarrow \forall t' : t' \geq t :: x(t') \leq 0) \wedge (x(t) \geq d \Rightarrow \forall t' : t' \geq t :: x(t') \geq d)$. Similarly, the "enter and cross the intersection" subgoal, g_{02}, is given by

$$\begin{aligned}
&(x(t) \leq 0 \Rightarrow \exists t_0 : t_0 \geq t :: (\forall t' : t' \geq t_0 :: x(t') \geq d))\\
&\wedge(x(t) \geq d \Rightarrow \exists t_0 : t_0 \geq t :: (\forall t' : t' \geq t_0 :: x(t') \leq 0))\\
&\wedge\{0 < x(t) < d \Rightarrow (\exists t_0 : t_0 \geq t :: (\forall t' : t' \geq t_0 :: x(t') \leq 0))\\
&\quad \vee(\exists t_0 : t_0 \geq t :: (\forall t' : t' \geq t_0 :: x(t') \geq d))\}.
\end{aligned}$$

The programs for subgoals g_{01} and g_{02} are shown in Figure 5(a) and Figure 5(b), respectively. In Figure 5(b), $p(t)$ is a function that computes the position of the car at the maximum acceleration at the end of the reaction time,$\delta(t)$, and $\alpha_{min}(t)$ returns the minimum acceleration needed to cross the intersection within the given reaction time. The union of the programs for the two subgoals can be derived automatically and is shown in Figure 5(c).

4 DEPENDABILITY ASSURANCE

The safety of each subsystem is assured independently by finding the set of unsafe states and ensuring that there are no transitions from any reachable states to the unsafe states.

Theorem 1. The safety of a disjunctive decomposition of $\mathbf{S_i}(t)$ into $\mathbf{S_{ij}}(t)$, $1 \leq j \leq n_i$, follows from the safety of its subsystems if the set of unsafe states in $\mathbf{S_{ij}}$, $1 \leq j \leq n_i$, is identical to the set of unsafe states in $\mathbf{S_i}(t)$.

Theorem 2. The safety of a conjunctive decomposition of $\mathbf{S}(t)$ into $\mathbf{S_i}(t)$, $1 \leq i \leq n$, follows from the safety of its subsystems if the subsystems are ranked in a priority order and, in the intersection operation, the more safety-critical version overrides a less critical version if the intersection is ϕ.

The stability of the individual subsystems can be analyzed by determining whether the goal states are reachable from all possible states, including failure states. Unlike other attributes, the stability of the system is more difficult to show in general.

Theorem 3. If P_1 and P_2 are stable IDEAL programs, then $P_1 \cap P_2$ is stable provided that there are no ϕ outputs.

For high-confidence certification, let s_i, $1 \leq i \leq n$, be the states in an RHFSM and let t_{ij}, $1 \leq i, j \leq n$, denote the transitions. To analyze the reliability of this HSFM, the following inputs are needed: (1) π_i, the probability of starting in state s_i, and p_{ij}, the conditional probability of selecting transition t_{ij} given that the state is s_i; these data must be provided by the domain expert; (2) r_{ij}, the reliability of each transition in the RHFSM, and c_i, the probability that state s_i has been correctly classified; these data are obtained from the analysis of the HFSM using the domain-specific analyzer. The reliability of the RHFSM is the probability that the selected trajectory (a) is achievable, (b) does not pass through any constraints, and (c) ends in a goal state. This is computed in the following way. Let T_i denote the set of trajectories to a goal state from state s_i.

1) Reliability of trajectory $x \in T_i$, $R(x) = \prod_{1 \leq j < |x|} c_{x(i)} \times r_{x(i)x(i+1)}$;
2) Probability of selecting trajectory $x \in T_i$, $p(x) = \prod_{1 \leq i < |x|} p_{x(i)x(i+1)}$;
3) Reliability of the RHFSM $= \sum_{i=1}^{n} \pi_i \sum_{x \in T_i} p(x)R(x)$.

The system level reliability follows from the subsystem level reliability in a simple way.

Theorem 4. If P_1 and P_2 are two IDEAL programs, then $R(P_1 \cap P_2) = R(P_1) \times R(P_2)$, provided P_1 and P_2 have independent failure processes.

Theorem 5. If P_{11} and P_{12} are two IDEAL programs, then $max\{R(P_{11}), R(P_{12})\} \geq R(P_{11} \cup P_{12}) \geq R(P_{11}) \times R(P_{12})$, provided P_1 and P_2 have independent failure processes.

Theorem 6. If P_{11}, P_{12}, and P_2 are three IDEAL programs, then $R((P_{11} \cup P_{12}) \cap P_2) = [1 - (1 - R(P_{11})) \times (1 - R(P_{12}))] \times R(P_2)$.

5 RELATED WORK

The research presented in this paper is related to rigorous automated software development methods and high-confidence assurance.

Our modeling approach is an extension of the synchronous paradigm [20] to handle time-varying state spaces [1] and relational programs [3, 4]. The use of relational programs allows complete isolation of the different RHFSMs; that is, each RHFSM is designed to be maximally compatible with any other RHFSM. Hence, a change to other RHFSMs, including the addition or removal of RHFSMs, does not affect a given RHFSM — it cannot be made any more compatible. (A limited form of multiple outputs is possible in PAISLey, but it is restricted to undefined functions and processes [21].)

Decomposing a state transition model into separate views is a crucial step in simplifying the model and making it amenable to analysis. One of the earliest works is reported in [21]. The concept of multiple views has also been used in Statecharts [11], Objectcharts [7], and RSML [17] among others and has been applied to existing languages, e.g. Z [15]. The novel feature in the IDEAL subsystem approach is that the model for each view is trajectory-oriented, i.e., it describes the set of all possible trajectories that can solve the control problem corresponding to the view. We achieve further simplicity by having no interaction between the views — all the views receive the same vector of sensor inputs and independently compute their trajectories.

Several types of assurance techniques have been developed for FSM-related models. Static analysis tools have been developed for checking nonfunctional properties, including the *completeness* [12, 13], *consistency* [12, 13], and *reachability* of the model [14] as well as various general assertions, such as absence of deadlocks and basic safety and liveness properties [14]. In our approach, the completeness analysis is simple since each RHFSM has the same state space as the overall system but with a subset of the goals and/or constraints; thus, completeness can be checked by showing that the disjunction of all the state predicates is equal to the overall state space. Similarly, consistency only needs to be checked for transitions having the same guard in two different RHFSMs. Model c hecking in the relational framework is more difficult. However, the underlying domain-specific knowledge base is an FSM (in fact, a cellular automaton for continuous control systems) and can be checked using SPIN [14].

For assurance of functional properties, it is necessary to have a model of the physical environment in addition to the specification of the control program. This was illustrated for a gas burner problem using duration calculus [6] in the ProCoS (Provably Correct Systems) project [19]. A more complex case study is reported in [10] where properties of the environment were formalized as axioms using PVS [8] for verifying safety properties of an aircraft wing and flap ontrol program. While these t ypes of checks and proofs enhance the confidence in the correctness of the model, it is difficult to quantify the confidence level. A variety of software reliability models have been developed over the past 25 years [18]. However, in addition to being controversial for safety-critical systems [5], there is no sound way of deducing the reliability of a system from the reliabilit y of its subsystems. The reason is that statistical reliability measures depend on the operational profile. It is difficult to assess the operational profile at the system lev el and impossible to do so at the subsystem lev el because of the absence of historical data. The beauty of composing IDEAL subsystems is that *every* subsystem (RHFSM) has the *same* state space, except for differences in goals, constraints, and free states; hence, every subsystem sees the same operational profile. Hence, the IDEAL subsystem approac h has the importan t (and unique) property that the subsystem reliability estimates can be statistically com bined to obtain the system reliability. Further, each RHFSM has relativ ely few states and transitions, which makes the reliability analysis feasible [9].

6 SUMMARY

In this paper, we have presented a method of decomposing an application in to subsystems such that the properties of the system can be deduced mathematicall y from the properties of its subsystems. Other adv antages of a system of IDEAL subsystems include (a) higher confidence levels since each IDEAL subsystem is m uch simpler than the entire application, (b) only simple axioms and local proofs and transformations are needed to compose the subsystems together, (c) faulty subsystems can be identified directly by analyzing their outputs and repair/recovery actions can also be confined within each subsystem separately, (d) facilitates multiparadigm implemen tations since the subsystems can be composed dynamically , (e) enables system evolution since there is a one-to-one, end-user visible correspondence between the requirements and the IDEAL subsystems, (f) high safety assurance since the priority scheme guaran tee that safety-critical subsystems will never be affected by faults in the other subsystems, and (g) facilitates dynamic composition, i.e., the subsystems can be executed concurren tly. Some future research areas include development of domain-specific methodologies for ac hieving designs that consist of IDEAL subsystems as well as more detailed assurance techniques.

Ac knowledgmen t

This research was supported in part by the National Science Foundation under Grant No. CCR-9900922, by the Texas ATP under Grant No. 009741-0143-1999, and by Alcatel USA and Texas Instrumen ts, Inc.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, “The algorithmic analysis of hybrid systems,” *Theo. Comp. Sc.*, Vol. 138, 1995, pp. 3 -34.

[2] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, Addison-W esley & ACM Press, New York, NY, 1991.

[3] F.B. Bastani, “Relational programs: An architecture for robust process-control programs,” Ann. of Software Engineering, Vol. 7, 1999, pp. 5-24.

[4] F.B. Bastani, V. Reddy, P. Srigiriraju, and I.-L. Yen, “A relational program architecture for the Bay Area Rapid Transit System,” Conf. on High Integrity Systems, Albuquerque, New Mexico, No v. 1999.

[5] R.W. Butler and G.B. Finelli, “The infeasibilit y of quantifying the reliability of life-critical real-time software,” *IEEE Trans. Softw. Eng.*, Vol. 16, No. 2, Feb. 1990, pp. 238-247.

[6] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn, "A calculus of durations," *Info. Proc. Lett.*, Vol. 40, No. 5, 1991, pp. 269-276.

[7] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to use Statecharts in object-oriented design," *IEEE Trans. on Softw. Eng.*, Vol. 18, No. 1, Jan. 1992, pp. 9-18.

[8] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial introduction to PVS," *Proc. Work. on Indus.-Str. Formal Spec. Tech.*, Apr. 1995.

[9] B. Cukic and F.B. Bastani, "On reducing the sensitivity of software reliability to variations in the operational profile," *IEEE Intl. Symp. on Softw. Rel. Eng.*, White Plains, NY, Oct. 1996.

[10] B. Dutertre and V. Stavridou, "Formal requirements analysis of an avionics control system," *IEEE Trans. on Softw. Eng.*, Vol. 23, No. 5, May 1997, pp. 267-278.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. of Comput. Prog.*, Vol. 8, 1987, pp. 231-274.

[12] M.P.E. Heimdahl and N.G. Leveson, "Completeness and consistency in hierarchical state-bed requirements," *IEEE Trans. Softw. Eng.*, Vol. 22, No. 6, June 1996, pp. 363-376.

[13] C.L. Heitmeyer, B.L. Labaw, and D. Kiskis, "Consistency checking of SCR-style requirements specifications," *Proc. 2nd IEEE Intl. Symp. Req. Eng.*, York, England,Mar. 1995, pp. 56-63.

[14] G.J. Holzmann, "The model checker SPIN,".*IEEE Trans. on Softw. Eng.*, Vol. 23, No. 5, May 1997, pp. 279-295.

[15] D. Jackson, "Structuring Z specifications with views," *ACM Trans. Softw. Eng. and Meth.*, Vol. 4, No. 4, Oct. 1995, pp. 365-389.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loigtier, J. Irwin, "Aspect-Oriented Programming," *Prof. European Conf. on Object-Oriented Programming (ECOOP)*, Finland, June 1997.

[17] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements specification for process-control systems," *IEEE Trans. on Softw. Eng.*, Vol. 20, No. 9, Sep. 1994, pp. 684-707.

[18] M. Lyu, (Ed.), *Handbook of Software Reliability Engineering,* McGraw-Hill and IEEE Comp. Soc. Press, 1996.

[19] A.P. Ravn, H. Rischel, and K.M. Hansen, "Specifying and verifying requirements of real-time systems," *IEEE Trans. on Softw.Eng.*, Vol. 19, No. 1, Jan. 1993, pp. 41-55.

[20] V. L. Winter and J. M. Boyle, "Proving refinement transformations for deriving high-assurance software," *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*, Oct. 1996.

[21] P. Zave, "A distributed alternative to Finite-State-Machine specifications," *ACM Trans. on Prog. Lang. and Sys.*, Vol. 7, No. 1, Jan. 1985, pp. 10-36.

OCL Component Invariants*

Hubert Baumeister Rolf Hennicker Alexander Knapp Martin Wirsing

Ludwig–Maximilians–Universität München
{baumeist, hennicke, knapp, wirsing}@informatik.uni-muenchen.de

Abstract

The "Object Constraint Language" (OCL) offers a formal notation for constraining model elements in UML diagrams. OCL consists of a navigational expression language which, for instance, can be used to state invariants and pre- and post-conditions in class diagrams. We discuss some problems in ensuring non-local, navigating OCL class invariants, as for bidirectional associations, in programming language implementations of UML diagrams, like in Java. As a remedy, we propose a component-based system specification method for using OCL constraints, distinguishing between global component invariants and local class invariants.

1 Introduction

During the last years the "Unified Modeling Language" (UML [2]) has become the de facto standard for object-oriented software development. The "Object Constraint Language" (OCL [14]) offers a formal notation to constrain the interpretation of model elements occurring in UML diagrams and therefore lends itself for systematic use in rigorous, UML-based software development methods, as shown, for example, in the Catalysis approach [5].

The OCL notation is particularly suited to constrain class diagrams since OCL expressions allow one to navigate along associations and to describe conditions on object states in class invariants and pre- and post-conditions of operations. However, by using the ability of describing navigational paths, a class invariant may be non-local in the sense that it also requires properties from other "remote" classes. This expressiveness and flexibility is appropriate in requirements specifications where the developer generally prefers a global view of the properties of the relationships between different classes. For design and implementation, however, such global requirements can be harmful since the implementation of a "remote" class would have to respect the non-local invariant of another class which is not mentioned anywhere in the "remote" class. Thus a programmer may not only have to check the validity of the invariant of the class he is implementing, but also the validity of invariants of other classes.

We first illustrate these problems with non-local class-based OCL invariants by simple examples, including the conventional use of "setter" operations and, more interestingly, standard OCL formalizations and Java implementations of bidirectional associations. As a remedy, we propose a component-based approach which has the following two properties: it allows us to write non-local invariants at the global level of components instead of at the local level of classes and it allows us to control the visibility of operations. An operation can be *component public* and therefore visible for all classes *inside and outside the component*; or an operation can be *component private* and therefore visible for all classes *inside the component*; or an operation can be *class private* and therefore visible only for its *own class*. Non-local invariants have to be respected only by component public operations; local invariants have to be respected by component public and component private operations; class private operations do not have to respect any invariant. However, for simplicity, we omit component hierarchy aspects and inheritance between different components.

In Sect. 2 we describe the problems with non-local class-based OCL invariants. In Sect. 3 we propose our component-based approach. In Sect. 4 we discuss how components can be realized in Java and we show some properties of correct realizations in Sect. 5. Throughout the paper we assume that the reader is familiar with UML class diagrams and the OCL notation.

2 Non-Local Class Invariants

For exhibiting the problems with non-local class-based invariants, we model a simple seminar system inspired by a similar example in the Catalysis book [5, Sect. 2.5.1, p. 67], see Fig. 1. In this system a course consists of several sessions. Each session has at most one instructor and each instructor may be qualified for several courses. Each session has a start and an end time. A session may also be public, where external participants have to pay a certain amount in order to be admitted. There are three invariants: the simple invariant for the class Session requires that the start time is before the end time; the invariant for class PublicSession states that the price for such a session is at least 10$. The invariant for the class Instructor requires that an instructor should only teach sessions for courses he is qualified for. The class Session shows an initialization operation setup setting default start and end times for a session; this operation is overridden in PublicSes-

*Partially supported by the DFG project InOpSys, ref. WI 841/6-1.

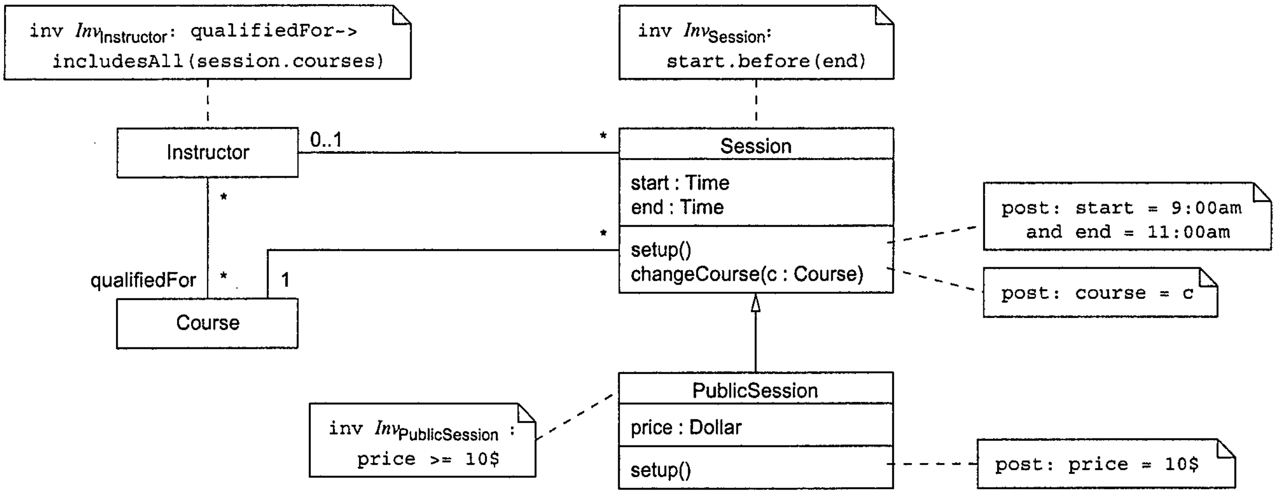

inv $Inv_{\mathsf{Instructor}}$: qualifiedFor-> includesAll(session.courses)
inv Inv_{Session}: start.before(end)
Instructor
0..1
*
Session
start : Time
end : Time
setup()
changeCourse(c : Course)
post: start = 9:00am and end = 11:00am
post: course = c
*
*
qualifiedFor
*
1
Course
PublicSession
price : Dollar
setup()
inv $Inv_{\mathsf{PublicSession}}$: price >= 10$
post: price = 10$

Figure 1: Annotated class diagram for the seminar example

sion initializing additionally the price for the public session. Moreover, the class Session has a "setter" operation changeCourse which allows to assign a new course to a session; the post-condition just requires to reassign the new course to the actual session.

For a correct implementation of this system in Java, one would like to require that any operation $C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)$ of any class C of the diagram preserves the invariant INV_C, obtained by the conjunction of the class invariant of C and the invariants of all its super-classes (if any), and satisfies the pre- and post-condition provided for op. As formalized in [12], this means that the Hoare formula

$$\{Pre_{C::op}\ \texttt{and}\ INV_C\}\quad C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)\quad \{Post_{C::op}\ \texttt{and}\ INV_C\}$$

should be valid. In the example, any implementation of the operation changeCourse of class Session should satisfy the Hoare formula

```
{start.before(end)}
  Session :: changeCourse(c : Course)          (∗)
{course = c and start.before(end)}
```

(for the implicit requirements of the bidirectional associations see below). The following Java implementation satisfies (∗):

```
void changeCourse(Course c) {
  course = c;
}
```

The problem is that, although changeCourse does not involve any attribute or role of class Instructor, it may destroy the invariant $Inv_{\mathsf{Instructor}}$ of class Instructor, e.g., when being called via s.changeCourse(c) for a session s having instructor s.instructor who is not qualified for the course c.

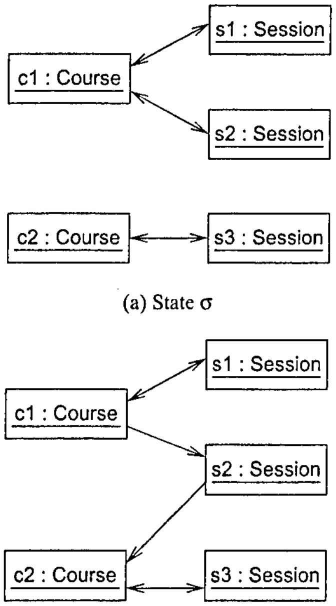
Another problem stems from making explicit the semantic constraints of bidirectional associations by expressing them in OCL. Consider for example the one-to-many association between the class Course and the class Session. The semantic constraint requires that any object c of class Course is related to a set of objects of class Session in such a way that each of these objects is related to c; thus navigating from c to any object of Session and back to class Course yields the original object c. Similarly, the sessions of the course of a Session object s must include the original object s. In OCL, one may try to formalize this using the following two class invariants of Session and Course:

```
context Course
  inv Inv'Course :
    self.session->
      forAll(s | s.course = self)

context Session
  inv Inv'Session :
    self.course.session->includes(self)
```

Now consider a system state σ showing two objects c_1, c_2 of class Course and three objects s_1, s_2, s_3 of class Session such that object c_1 is related with objects s_1 and s_2, and object c_2 is related with object s_3, see Fig. 2(a). Obviously, a Java call s_2.changeCourse(c_2) does not respect the invariants $Inv'_{\mathsf{Course}}[c_1/\texttt{self}]$ and $Inv'_{\mathsf{Session}}[s_2/\texttt{self}]$, cf. the object diagram in Fig. 2(b).

According to Hitz and Kappel [7, Sect. 6.2.1, p. 271–275], a correct Java implementation of changeCourse with respect to the bidirectional association can be given using two Java operations addSession and rmSession in the following way:

```
public class Session {
  private Instructor instructor;
  private Course course;

  public void changeCourse(Course c) {
    if (course != c) {
      if (course != null)
```



(a) State σ

(b) State after s_2.changeCourse(c_2)

Figure 2: Sample states of the seminar system

```
        course.rmSession(this);
      course = c;
      c.addSession(this);
    }
  }
  ...
}

public class Course {
  private Vector session;

  public void addSession(Session s) {
    if (!session.contains(s))) {
      session.addElement(s);
      s.changeCourse(this);
    }
  }

  public void rmSession(Session s) {
    session.removeElement(s);
  }

  ...
}
```

The operations changeCourse and addSession indeed preserve both of the invariants Inv'_{Course} and Inv'_{Session} but (as Hitz and Kappel also mention in their book [7]) calling rmSession may lead to an illegal state; e.g., see Fig. 3, calling c_2.rmSession(s_3) in state σ leads to a state where the invariant $Inv'_{\mathsf{Session}}[s_3/\mathtt{self}]$ does not hold.

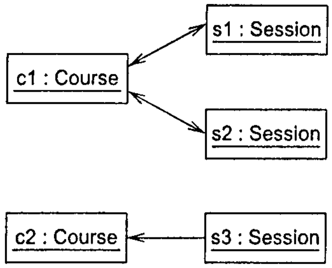


Figure 3: State after c_2.rmSession(s_3)

3 Component-Based Invariants

The problems described in the previous section are due to the fact that class invariants of one class are based on properties of objects of another class; in other words, the invariant expressions navigate to objects of other classes. For instance, the invariant $Inv_{\mathsf{Instructor}}$ navigates to the sessions of an instructor and requires that all these sessions are for courses the instructor is qualified for. Hence it is obvious that the class invariant for instructor can easily be destroyed by changing the course of a session. To overcome these problems we use a component-oriented development methodology. Component-based approaches for software development have been advocated by many authors including Broy [3] and Szyperski [13], or, in the context of UML and OCL, by Catalysis [5] and Cheesman and Daniels [4].

We do not propose a new notion of component; almost any of the notions for components in the literature is suitable for our approach provided that a component is composed of classes (and possibly local components) and that the following two requirements are satisfied:

1. It is possible to require invariants globally for the whole component and also locally for the elements of a component.
2. An operation can be declared to be visible either inside and outside the component, or only inside the component, or only inside a single class of the component.

More precisely, we distinguish between class invariants and component invariants: A *class invariant* is an invariant for describing properties concerning a single class (i.e. its attributes and association roles without navigation) and a *component invariant* is an invariant for describing properties concerning two or more classes. For example, the invariant $Inv_{\mathsf{Instructor}}$ of class Instructor cannot be used as a class invariant but has to be included in the component invariant for the seminar system (cf. Fig. 4). The invariants induced by bidirectional associations are also included in the component invariant Inv_{Seminar}. The invariants Inv_{Session} and $Inv_{\mathsf{PublicSession}}$ of classes Session and PublicSession, however, are class invariants.

Concerning the visibility of operations we distinguish between operations which are

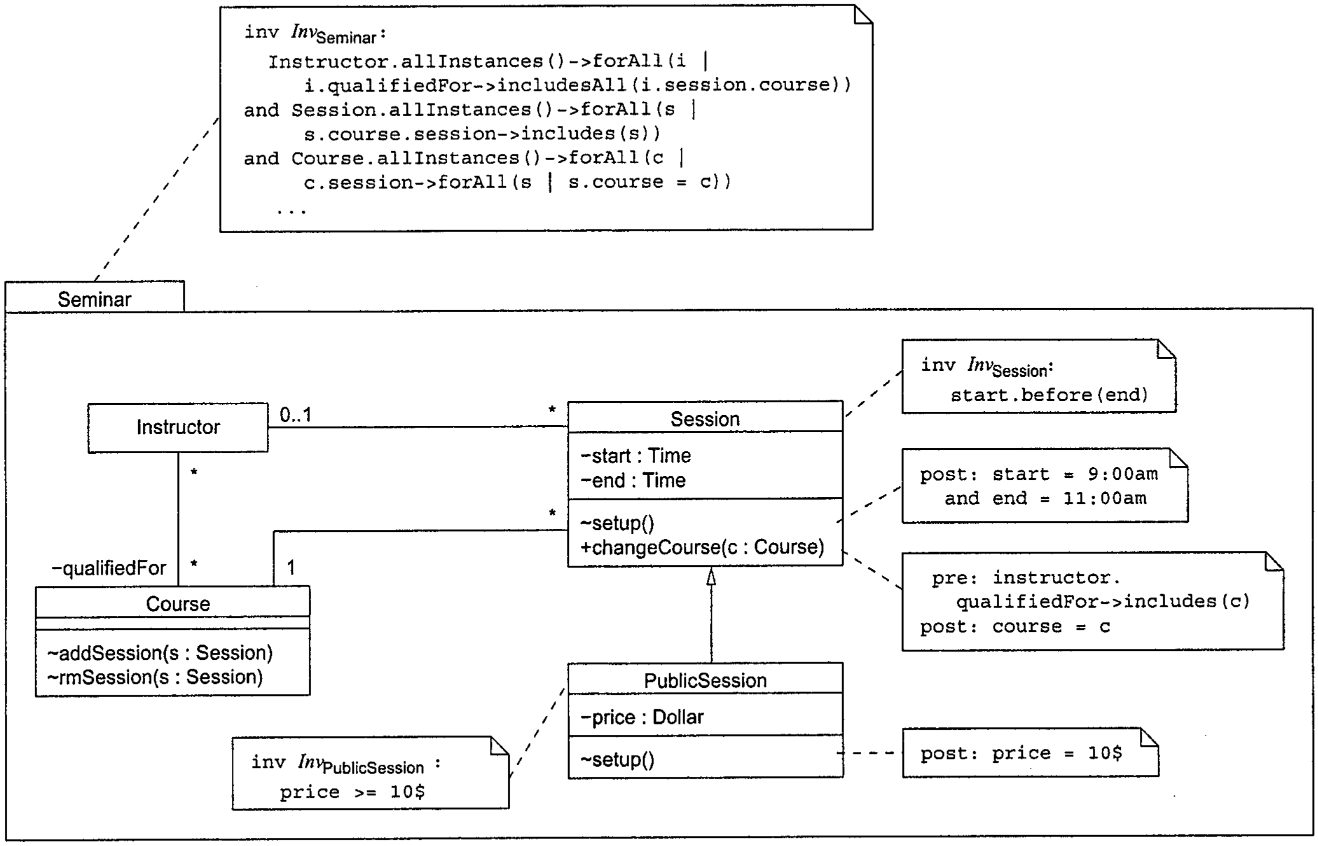

inv Inv_{Seminar}:
Instructor.allInstances()->forAll(i |
i.qualifiedFor->includesAll(i.session.course))
and Session.allInstances()->forAll(s |
s.course.session->includes(s))
and Course.allInstances()->forAll(c |
c.session->forAll(s | s.course = c))
...
Seminar
Instructor
0..1
*
Session
-start : Time
-end : Time
~setup()
+changeCourse(c : Course)
inv Inv_{Session}:
start.before(end)
post: start = 9:00am
and end = 11:00am
pre: instructor.
qualifiedFor->includes(c)
post: course = c
*
-qualifiedFor
*
1
*
Course
~addSession(s : Session)
~rmSession(s : Session)
PublicSession
-price : Dollar
~setup()
inv $Inv_{\text{PublicSession}}$:
price >= 10$
post: price = 10$

Figure 4: Component model of the seminar system

component public —
visible at the interface of the component
component private —
visible to all classes of the component
class private —
visible to a single class of the component

Each component public operation has to preserve the component invariant, the class invariant where the operation is declared, and the class invariants of all super-classes, and satisfy their pre-/post-condition. Each component private operation has to preserve its class invariant and the class invariants of all super-classes and satisfy its pre-/post-condition. Finally, each class private operation has to satisfy its pre-/post-condition. Since class private operations are auxiliary operations which can be (internally) applied to an object in a "non-stable" state, they need not preserve invariants; cf. also [9].

Concerning the visibility of attributes we assume a good style of design: all attributes have to be visible only to a single class.

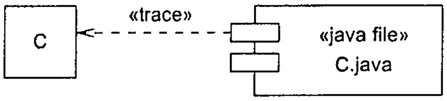
Components in the sense of Cheesman and Daniels, Catalysis, or UML subsystems can express these visibility requirements and also show notations for invariants. In particular, due to the explicit notion of interfaces used in Catalysis and by Cheesman and Daniels the visibility of component public operations can be modeled explicitly. A UML subsystem provides the following visibility correspondences for operations [11]: class private corresponds to private (–), component private to package (~), and component public to public (+). Similarly, an attribute that is only visible to a single class has visibility private (–).

Fig. 4 shows the seminar system as a UML subsystem component. We choose to declare rmSession and addSession not to be component public but only to be component private. Hence the component invariant Inv_{Seminar} needs not to be respected by these operations, but has to be respected by the component public operation changeCourse. In order to be able to fulfill the post-condition of changeCourse and, simultaneously, to ensure the preservation of the component invariant Inv_{Seminar}, we have to add a pre-condition to changeCourse, requiring the replacing course to be compatible with the abilities of the teaching instructor. In fact, the detection of this pre-condition is greatly facilitated by making the component invariant explicit.

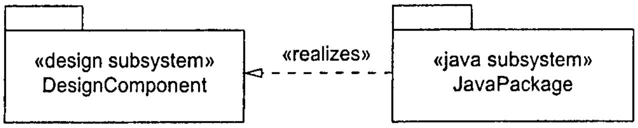
4 Realization of Components

Based on our notion of component we define a realization relation which connects a UML design component and a Java implementation model. As implementation model for components we use Java packages, classes in a design component are mapped to Java classes. However, we put an additional restriction on Java classes: `private` Java attributes and methods are only called on `this` which is good programming practice for encapsulating object states.

We employ UML trace and realization dependencies as considered in [1] to relate UML classes and Java classes, and UML design components and Java packages. A realization relation between a UML design component and a Java package expresses that the implementation model satisfies the requirements of the design model. In particular, the Java methods corresponding to a component public, component private, or class private operations in a class of the UML design component have to preserve the component invariant, the class invariant (and the class invariants of all super-classes), and satisfy its pre-/post-condition according to the requirements described in Sect. 3. Similarly, we have to consider component public, component private, and class private object constructors, where we always require that constructors establish class invariants. Trace dependencies guarantee that the OCL expressions used as constraints for the design model can be interpreted in the implementation model. The constraints on the Java implementation are represented by Hoare formulae and can be proved using the calculus presented in [12].



(a) Trace dependency



(b) Realization dependency

Figure 5: Component dependencies

A trace dependency holds between a UML design class C and a Java class C.`java`, see Fig. 5(a), if the direct super-classes of C and of C.`java` coincide, if the operations of C and the methods of C.`java` coincide (up to an obvious syntactic modification of the signature), if all attributes of C are also attributes of C.`java`, and if for each (explicit or implicit) role name at a navigable association end C.`java` contains a corresponding reference attribute with the same name. (Note that standard types may be slightly renamed according to the Java syntax and that role names with multiplicity greater than one map to reference attributes of some container type.) Concerning visibility the correspondences are as follows [6]: Component public operations correspond to public methods (of public classes) in Java, component private operations to Java default visible methods, and class private operations to private Java methods; private attributes in UML correspond to private attributes in Java.

A realization dependency holds between a UML design component M and a Java package P, see Fig. 5(b), if the following conditions are satisfied: Let Inv_M denote the component invariant of the design component M and let Inv_C denote the class invariant of class C in M. Let INV_C denote the inherited class invariant of class C, i.e., the conjunction of the class invariant of C and the class invariants of the super-classes of C:

$$INV_C = \bigwedge_{D \geq C} Inv_D$$

1. For all classes C in M there is exactly one class C.`java` in P such that C and C.`java` are related by a trace dependency.

2. Let op be an operation declared in the design class C with constraint

   $\texttt{context}\ C\texttt{::}op\texttt{(}x_1\ \texttt{:}\ D_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{:}\ D_n\texttt{)}$
   $\quad\texttt{pre:}\ Pre_{C::op}$
   $\quad\texttt{post:}\ Post_{C::op}$

   (a) If op is class private then its corresponding private method op in C.`java` satisfies the pre-/post-condition (but no invariants):

   $$\{Pre_{C::op}\}\ C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)\ \{Post_{C::op}\}$$

   (b) If op is component private then the corresponding default visible method op in C.`java` preserves the inherited class invariant INV_C and satisfies the pre-/post-condition:

   $$\{Pre_{C::op}\ \texttt{and}\ INV_C\}\ C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)\ \{Post_{C::op}\ \texttt{and}\ INV_C\}$$

   Moreover, any call of op for an object of a class D that is a sub-class of C, has to preserve the inherited invariant of D and satisfy the inherited pre-/post-condition for op:

   $$\{Pre_{C::op}\ \texttt{and}\ INV_D\}\ D :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)\ \{Post_{C::op}\ \texttt{and}\ INV_D\}$$

   (c) If op is component public then the corresponding public method op in C.`java` fulfills the requirements for component private operations and additionally preserves the component invariant Inv_M:

   $$\{Pre_{C::op}\ \texttt{and}\ INV_C\ \texttt{and}\ Inv_M\}\ C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)\ \{Inv_M\}$$

3. Let $C(x_1,\ldots,x_n)$ be a constructor of the design class C with constraint

$$\begin{array}{l}\texttt{context } C\texttt{::}C\texttt{(}x_1 \ \texttt{:}\ D_1\texttt{, ..., } x_n \ \texttt{:}\ D_n\texttt{)}\\ \quad \texttt{pre: } Pre_C\\ \quad \texttt{post: } Post_C\end{array}$$

   (a) If the constructor C is class private or component private then the corresponding private or default visible constructor C in C`.java` establishes the inherited class invariant INV_C and satisfies the pre-/post-condition:

$$\begin{array}{l}\{Pre_C\}\\ \quad x = \texttt{new } C\texttt{(}x_1 : D_1\texttt{, ..., } x_n : D_n\texttt{)}\\ \{Post_C[x/\texttt{self}] \texttt{ and } INV_C[x/\texttt{self}]\}\end{array}$$

   (b) If the constructor C is component public then the corresponding public constructor C in C`.java` fulfills the requirements for component private constructors and additionally preserves the component invariant Inv_M:

$$\begin{array}{l}\{Pre_C \texttt{ and } Inv_M\}\\ \quad x = \texttt{new } C\texttt{(}x_1 : D_1\texttt{, ..., } x_n : D_n\texttt{)}\\ \{Inv_M\}\end{array}$$

We say that a Java package is a correct realization of a UML design component M if a realization dependency holds between M and P. The requirements for a correct realization follow the conditions for class correctness in [9] or [8] which here are extended to take into account visibilities and component invariants. Note that, by requirement (2b) Liskov's substitution principle is satisfied with respect to pre-/post-conditions of inherited operations.

For the seminar system introduced in Sect. 2, we can establish a realization dependency between the UML design component in Fig. 4 and the Java implementation in Sect. 2 changing the `public` visibilities of `addSession` and `rmSession` into default visibilities. The proof obligations for `setup` originating from requirement (2b) are:

$$\begin{array}{l}\{\texttt{start.before(end)}\}\\ \quad \texttt{Session :: setup()}\\ \{\texttt{start = 9:00am and end = 11:00am and}\\ \qquad \texttt{start.before(end)}\}\end{array}$$

$$\begin{array}{l}\{\texttt{start.before(end) and price >= 10\$}\}\\ \quad \texttt{PublicSession :: setup()}\\ \{\texttt{start = 9:00am and end = 11:00am and}\\ \qquad \texttt{start.before(end) and price >= 10\$}\}\end{array}$$

$$\begin{array}{l}\{\texttt{start.before(end) and price >= 10\$}\}\\ \quad \texttt{PublicSession :: setup()}\\ \{\texttt{price = 10\$ and}\\ \qquad \texttt{start.before(end) and price >= 10\$}\}\end{array}$$

Obviously, the last two requirments can be easily combined into a single requirement.

Similarly, the proof obligations for `changeCourse` originating from requirement (2b) and (2c) are:

$$\begin{array}{l}\{\texttt{instructor.qualifiedFor->includes(c) and}\\ \qquad \texttt{start.before(end)}\}\\ \quad \texttt{Session :: changeCourse(c : Course)}\\ \{\texttt{course = c and start.before(end)}\}\end{array}$$

$$\begin{array}{l}\{\texttt{instructor.qualifiedFor->includes(c) and}\\ \qquad \texttt{start.before(end) and price >= 10\$}\}\\ \quad \texttt{PublicSession :: changeCourse(c : Course)}\\ \{\texttt{course = c and}\\ \qquad \texttt{start.before(end) and price >= 10\$}\}\end{array}$$

$$\begin{array}{l}\{\texttt{instructor.qualifiedFor->includes(c) and}\\ \qquad \texttt{start.before(end) and } Inv_{\mathsf{Seminar}}\}\\ \quad \texttt{Session :: changeCourse(c : Course)}\\ \{Inv_{\mathsf{Seminar}}\}\end{array}$$

5 Properties of Correct Component Realizations

In a correct Java realization of a UML design component, method and constructor calls cannot destroy the class invariants of alien objects, i.e. of any object different from the object the method is called upon or different from the newly created object; additionally, all objects created during the execution of an operation or a constructor satisfy their class invariants.

Lemma 1. *Consider a correct realization of a design model. Assume that for any terminating method call* $o\texttt{.}op\texttt{(}d_1\texttt{, ..., } d_n\texttt{)}$ *and any terminating constructor call* $o = \texttt{new } C\texttt{(}d'_1\texttt{, ..., } d'_m\texttt{)}$ *the pre-condition of any method called during the evaluation of* $o\texttt{.}op\texttt{(}d_1\texttt{, ..., } d_n\texttt{)}$ *and* $o = \texttt{new } C\texttt{(}d'_1\texttt{, ..., } d'_m\texttt{)}$ *is satisfied. Then for any classes C and D of the implementation model, object o of class C, method op of C, and object* $o' \neq o$ *of class D existing in the state after executing* $o\texttt{.}op\texttt{(}d_1\texttt{, ..., } d_n\texttt{)}$*:*

$$\begin{array}{l}\{Inv_D[o'/\texttt{self}] \texttt{ and}\\ \ D\texttt{.allInstances()->includes(}o'\texttt{)}\}\\ \quad o\texttt{.}op\texttt{(}d_1\texttt{, ..., } d_n\texttt{)}\\ \{Inv_D[o'/\texttt{self}]\}\end{array} \tag{1}$$

$$\begin{array}{l}\{\texttt{not } D\texttt{.allInstances()->includes(}o'\texttt{)}\}\\ \quad o\texttt{.}op\texttt{(}d_1\texttt{, ..., } d_n\texttt{)}\\ \{o'\texttt{.oclIsNew() and } Inv_D[o'/\texttt{self}]\}\end{array} \tag{2}$$

where $d_1,\ldots,d_n$ are some objects. Moreover, for any classes C and D of the implementation model, object o of class C, method op of C, and object $o' \neq o$ of class D existing in the state after executing o = new $C(d'_1, \ldots, d'_m)$*:*

$$\begin{array}{l}\{Inv_D[o'/\texttt{self}] \ \texttt{and} \\ \ D.\texttt{allInstances()->includes(}o'\texttt{)}\} \\ \quad o = \texttt{new } C\texttt{(}d'_1, \ldots, d'_m\texttt{)} \\ \{Inv_D[o'/\texttt{self}]\}\end{array} \tag{3}$$

$$\begin{array}{l}\{\texttt{not } D.\texttt{allInstances()->includes(}o'\texttt{)}\} \\ \quad o = \texttt{new } C\texttt{(}d'_1, \ldots, d'_m\texttt{)} \\ \{o'.\texttt{oclIsNew()} \ \texttt{and} \ Inv_D[o'/\texttt{self}]\}\end{array} \tag{4}$$

where $d'_1,\ldots,d'_m$ are some objects.

Proof sketch. The claims are proved simultaneously by induction on the depth of the execution tree of method calls and object creations.

Each method call and each object creation is associated with an execution trace consisting of attribute assignments $o.f$ = v, method calls $o.op(v_1, \ldots, v_l)$, and object creations o = new $C(v_1, \ldots, v_l)$.

We define the degree of a method call, an object creation, and an attribute assignment as follows: $\deg(o.f = v) = 0$, $\deg(o.op(d_1, \ldots, d_n)) = 1 + \max_{1 \leq i \leq k}\{\deg(s_i)\}$ where $s_1 \ldots s_k$ is the execution trace of $o.op(d_1, \ldots, d_n)$, and $\deg(o = \texttt{new } C(v_1, \ldots, v_l)) = 1 + \max_{1 \leq i \leq k}\{\deg(s_i)\}$ where $s_1 \ldots s_k$ is the execution trace of o = new $C(v_1, \ldots, v_l)$.

Let the assumptions of the claim hold and let o' be as in the claim. For any method call and any object creation, we proof the claim by induction on the degree of the method call or object creation and by sub-induction on the length of the execution trace.

Case 1. Let $s_1 \ldots s_k$ be the execution trace of a method call $o.op(d_1, \ldots, d_n)$. Let the pre-condition $Inv_D[o'/\texttt{self}]$ and $D.\texttt{allInstances()->includes(}o'\texttt{)}$ hold before execution of $s_1 \ldots s_k$.

Case 1.1. Let $\deg(o.op(d_1, \ldots, d_n)) = 1$. If $k = 0$ then $Inv_D[o'/\texttt{self}]$ holds trivially after $s_1 \ldots s_k$. Let $k > 0$ and let $Inv_D[o'/\texttt{self}]$ hold before the execution of s_k. Then s_k has necessarily the form $o.f$ = v, since all attributes have to be private and attributes are only called on this. Hence, $Inv_D[o'/\texttt{self}]$ holds after the execution of s_k since $o' \neq o$ and all class invariants have to be local, i.e., they must not employ navigation beyond the scope of a single object.

Case 1.2. Let $\deg(o.op(d_1, \ldots, d_n)) = m + 1$. If $k = 0$ then $Inv_D[o'/\texttt{self}]$ holds trivially after $s_1 \ldots s_k$. Let $k > 0$ and let $Inv_D[o'/\texttt{self}]$ hold before the execution of s_k. Then there are three cases:

Case 1.2.1. If $s_k = o.f$ = v, then the same argument as in case 1.1 applies.

Case 1.2.2. Let $s_k = o''.op'(v_1, \ldots, v_l)$. If $o'' = o'$ then op' cannot be private by our restriction on the programming style of Java classes. Hence, by proof obligations (2b–c) and the pre-condition assumption, $Inv_D[o'/\texttt{self}]$ holds after execution of s_k. If $o'' \neq o'$ then $Inv_D[o'/\texttt{self}]$ holds after execution of s_k by the main induction hypothesis, since $\deg(s_k) \leq m$.

Case 1.2.3. Let $s_k = o''$ = new $C'(v_1, \ldots, v_l)$. Then $o'' \neq o'$. Thus, $Inv_D[o'/\texttt{self}]$ holds after execution of s_k by the main induction hypothesis, since, again, $\deg(s_k) \leq m$.

An analogous argument holds for all remaining cases. □

From this observation, it follows that in a correct Java realization of a UML design model, any non-class private operation preserves all class invariants and, moreover, any component public operation preserves all invariants including the component invariant. In order to prove these properties let INV_M denote the environment invariant of design component M, i.e., the conjunction of all class invariants of the classes in M for all instances:

$$INV_M = \bigwedge_C C.\texttt{allInstances()->} \\ \texttt{forAll(x | } Inv_C[\texttt{x/self}]\texttt{)}$$

Theorem 2. *Consider a correct realization of a design model M. Assume that for any terminating method call* $o.op(d_1, \ldots, d_n)$ *and any terminating constructor call* o = new $C(d'_1, \ldots, d'_m)$ *the pre-condition of any method called during the evaluation of* $o.op(d_1, \ldots, d_n)$ *and* o = new $C(d'_1, \ldots, d'_m)$ *is satisfied. Then for any class C of the implementation model and any non-class private method op of C*

$$\begin{array}{l}\{Prec_{C::op} \ \texttt{and} \ INV_M\} \\ \quad C :: op(x_1 : D_1, \ldots, x_n : D_n) \\ \{INV_M\}\end{array}$$

holds; moreover, for any constructor $C(x_1,\ldots,x_n)$ of C

$$\begin{array}{l}\{Prec_{C::C} \ \texttt{and} \ INV_M\} \\ \quad x = \texttt{new } C(x_1 : D_1, \ldots, x_n : D_n) \\ \{INV_M\}\end{array}$$

holds.

Proof. Let $o.op(d_1, \ldots, d_n)$ be a terminating call of a non-class private method *op* of class C of the implementation model. Let $Inv_D[o'/\texttt{self}]$ be any class invariant of INV_M where o' is an object existing in the state after executing $o.op(d_1, \ldots, d_n)$. If $o' = o$ and thus $D \leq C$, then $Inv_D[o'/\texttt{self}]$ is ensured by the assumptions (2b) on preservation of invariants of non-class private methods in correct realizations of the design model. Otherwise, i.e. if $o' \neq o$, apply Lemma 1.

The claim on constructors is proved analogously. □

Corollary 3. *With the assumption as in the theorem, any call of a component public operation or constructor preserves all* *the invariants, i.e.*

$$\{Prec_{C::op} \text{ and } INV_M \text{ and } Inv_M\}$$
$$C :: op\,(x_1 : D_1,\ \ldots,\ x_n : D_n)$$
$$\{INV_M \text{ and } Inv_M\}$$

$$\{Prec_{C::C} \text{ and } INV_M \text{ and } Inv_M\}$$
$$x = \text{new } C\,(x_1 : D_1,\ \ldots,\ x_n : D_n)$$
$$\{INV_M \text{ and } Inv_M\}$$

6 Conclusions

We have emphasized that pure class diagram-based object-oriented software development has some drawbacks related to invariants which can be overcome by using a component-based approach. Of course, the problems presented here are not the only problems in object-oriented and component-based software development. In particular, we have omitted a discussion of component hierarchies and associations between components, which may be treated by repeating the component invariant approach at all hierarchy levels. Inheritance between an element of one component and an element of another component seems to pose some more subtle problems. For instance, the interplay between inherited operations and component invariant preservation is not obvious. The issue of sharing between components [10] may have similar effects.

However, OCL has proven to be a valuable tool for analyzing well-known implementation schemata for associations between classes. In our opinion, OCL is well-suited as a constraint language for UML and presents a further positive step towards rigorous object-oriented software development.

References

[1] M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct Realizations of Interface Constraints with OCL. In R. B. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 399–415. Springer, Berlin, 1999.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison–Wesley, Reading, Mass., &c., 1998.

[3] M. Broy. Towards a Mathematical Concept of a Component and its Use. *Software — Concepts and Tools*, 18:137–148, 1997.

[4] J. Cheesman and J. Daniels. *UML Components.* Addison Wesley, Boston, &c., 2000.

[5] D. F. D'Souza and A. C. Wills. *Object, Components, Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, Mass., &c., 1998.

[6] J. Gosling, B. Joy, G. Steele, and G. Brancha. *The Java Language Specification.* Addison–Wesley, Reading, Mass., &c., 2nd edition, 2000.

[7] M. Hitz and G. Kappel. *UML@Work.* dpunkt.verlag, Heidelberg, 1999.

[8] K. Huizing and R. Kuiper. Verification of Object-Oriented Programs Using Class Invariants. In T. Maibaum, editor, *Proc. Intl. Conf. Fundamental Approaches to Software Engineering 2000*, volume 1783 of *Lect. Notes Comp. Sci.*, Berlin, 2000. Springer, Berlin.

[9] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, New York, &c., 1988.

[10] P. Müller and A. Poetzsch-Heffter. Modular Specification and Verification Techniques for Object-Oriented Software Components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems.* Cambridge University Press, Cambridge, 2000.

[11] Object Management Group. Unified Modeling Language Specification, Version 1.4. Draft, OMG, 2001. `http://cgi.omg.org/cgi-bin/doc?ad/01-02-14`.

[12] B. Reus, M. Wirsing, and R. Hennicker. A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In H. Hußmann, editor, *Proc. 4th Int. Conf. Fundamental Approaches to Software Engineering,* volume 2029 of *Lect. Notes Comp. Sci.*, pages 300–317. Springer, Berlin, 2001.

[13] C. Szyperski. *Component Software.* Addison-Wesley, Harlow, &c., 1998.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison–Wesley, Reading, Mass., &c., 1999.

XML types are parsers

Peter T. Breuer   Carlos Delgado Kloos   Luis Sánchez Fernández
Ma. Carmen Fernández Panadero   Andres Marín López

Depto. Ingeniería Telématica
Universidad Carlos III de Madrid
Av. Universidàd, 30
E-28911 Leganés (Madrid/Spain)

E-mail: {ptb,cdk,luis,mcfp,amarin}@it.uc3m.es

Abstract

Phil Wadler claims in [1] that XML *has types*. In this article, we claim that XML *types are parsers*, and that equality of types is the functional equality of parsers, and not the formal syntactic identity between types. A system of sound logical rules for reasoning about inclusion between types is presented, and an algorithm for calculating a large subrelation of the inclusion relation is put forward. Arguments for the relative completeness of the rule system and the algorithm are described.

1 XML schemas as types

In an interesting document [1] produced by Phil Wadler of the W3C committee examining XML and XSL, the point is made that XML *has types*, and that document definitions and XML schemas can be likened to type specifications. Whether or not the details given there are correct, the important observation is that a type system for XML is possible. The idea is that a document tag pair "<a> ... </a>" can be seen as a data constructor. That is, at the abstract level, the XML scheme implicitly defines a data type, that we shall call a, and an associated unique outfix constructor that we shall denote $\langle \mathsf{a}\rangle \ldots \langle/\mathsf{a}\rangle$.

In order to make this observation, Wadler makes two preliminary observations. The first is the familiar *XML documents are trees*. But what kind of trees exactly? The answer is: many different kinds of *n-ary* trees, with the tags serving to denote which kind is which. At the abstract level, the tag identifiers correspond to labels on the nodes of the trees. The *contents* found between the document tag pairs correspond to the branches of the tree. The following data type definition is suitable for the abstract tree representation of a document:

$$t = \langle a\rangle\, e_1\, \ldots\, \langle/a\rangle \qquad (1)$$

$$e = t \mid s \mid \ldots \qquad (2)$$

where $t \in$ Tree, $e, e_1, \ldots \in$ Element, s is a string, and a is a tag identifier. This is an n-ary tree: a label and a list of branches. XML admits other atomic types than strings, but they will not be considered here.

But how does one represent the document tag attributes and other headers? This is Wadler's other observation, one well known to those with experience of XML: attributes in an opening tag may be regarded as merely the first of the elements inside the document tag pair. A tag with an attribute

```
<a foo="bar"> ... </a>
```

can be regarded as shorthand for a tag pair containing a new structure as the first of its sequence of document elements:

```
<a> <a.foo> bar </a.foo> ... </a>
```

The tag identifier a becomes the label on the topmost node in the abstract tree representation, and its first branch is a node with the label `a.foo`, while `bar` is a leaf string (quotes around strings are not necessary when the string occurs between tags instead of inside them).

Thus the type of document with an attributed tag corresponds to a double data type (and data constructor) definition, in which the attribute forms the first element within the outer tag:

$$\begin{aligned} \texttt{a} &::= \langle \mathrm{a}\rangle\, \texttt{a.foo} \ldots \langle /\mathrm{a}\rangle \\ \texttt{a.foo} &::= \langle \mathrm{a.foo}\rangle\, \texttt{String}\ \langle /\mathrm{a.foo}\rangle \end{aligned}$$

This is correct, but not the end of the story. Other kinds of elements may appear, such as sequences. The whole content sequence within a tag pair itself has a sequence type. Unfortunately Wadler chooses product types to represent this type of sequences of elements, and this is not correct. According to Wadler, the type of the contents of the tag $\langle \mathrm{a}\rangle \ldots \langle /\mathrm{a}\rangle$ is:

$$(\ \texttt{a.foo String},\ \ldots\ )$$

That is, a product type. This is not correct because the type product (x, x^*) – pairs of x and lists of x – is not the same as (x^*, x) or (x^*) as product types, and the intention of XML schema is to regard the document types defined by tagged structures with these variantly typed contents as equally typed.

The confusion extends also to elements which are themselves sequences. We may require an element to be a sequence of sequences of strings, for example. This is the same in XML as requiring it to be a simple sequence of strings, but standard type semantics will differentiate between lists of lists of strings `String**` and lists of strings `String*`. So standard types are not right for XML. We want a product type that more closely corresponds to the semantics of uni-level sequences, such as the *bunches* discussed in [3].

In the rest of this section a format and logic for XML types is presented that works its way around the problem noted above, and which provides a solid formal substrate for the interpretation and proper comprehension of XML document types. Then in Section 2 we will give an interpretation for types as parsers, thus the title of this article: *XML types are parsers.* XML schemas will be seen to correspond to parsers, and the construction of the parsers will be directed by the type. The resulting parsers satisfy all the semantic relations that we want, including, for example $(x, (y, z)) = ((x, y), z)$ and $(x, x^*) \leq x^*$. First, however, we describe XML in general terms, taking it from two different points of view in turn: the relation between XML and HTML and SGML; and XML as a group of related language, library and translation specifications, introducing the terminology of an XML document.

1.1 What is XML?

XML can be understood in various ways. It is a metalanguage, and also a set of standards.

1.1.1 XML as a metalanguage

XML (Extensible Markup Language) [8] is a simplification of SGML (Standard Generalized Makup Language). Both XML and HTML have emerged from SGML. SGML is also a meta-language — in other words, a language that serves to define other languages. While HTML is an application of SGML, XML is not an SGML application itself but rather a reduced version of SGML that is aimed at the world wide web.

Although the simplicity of HTML initially appeared as an advantage, with the passing of time it has become apparent that there are important requirements which such a simple language is not able to satisfy. HTML is not extensible. That is to say, one cannot just add new tags as necessary, but instead one must wait until the following version of the standard for a tag's possible inclusion, and then browsers may begin to support the extension.

XML emerged as an intermediate solution between SGML and HTML. In XML, as compared to SGML, there are some restrictions: the punctuation in the lexicon is already fixed and there are limitations on the syntax (for example, all tags should be opened and closed in a balanced fashion — with the exception of empty tags, which have a special form).

The flexibility remaining in the syntax permits the creation of different languages. One may specify the set of tags and the restrictions on grouping them. This specification always appears in a separate document, called a DTD (Document Type Definition) document. It is said that XML conserves 80% of the flexibility of SGML and only has 20% of its complexity. In addition, XML has been specially designed for the web.

1.1.2 XML as a set of specifications

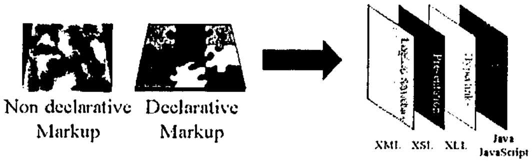
XML is not only a metalanguage for the definition of personalized languages, it is a family of standards which include all the related aspects of a document: its logical structure, its presentation, its hypertext structure, etc.

XML 1.0 is the standard which defines the format for specifying the logical structure of a document. XSL (Extensible Stylesheet Language) [9] is the standard that describes the shape of a XML document. It has two parts: XSLT [10], which describes how to transform one XML document to another; XSL-FO, which defines a vocabulary with which to specify the format of a XML document.

The family of standards connected with the hypertext qualities of a document is called XLL (Extensible Linking Language). Amogst them are XLink [6] and XPointer [4]. XLink defines the form in which a reference is made to a document that we want to link to while XPointer defines how the targets are marked in said document.

To understand in a little more detail the relation between the different standards, a document should be considered as a complex structure of various parts, as illustrated in figure 1:

- logically structured information (XML);
- physical layout information (XSL);
- relations between different parts of the document or other documents (XLL).
- response of the document to determined events (eg. interaction with the user).



- XML: Information exchange
- XML+XSL: Publishing
- XML+XSL+XLL: Navigation
- XML+XSL+XLL+ script: Application

Figure 1: Document Structure

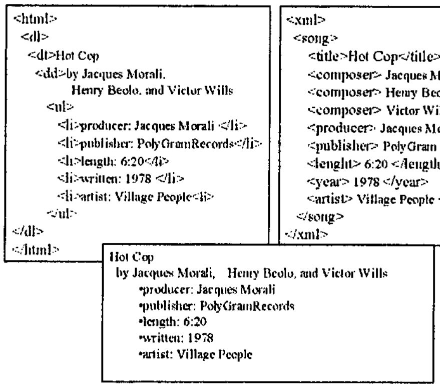
XML can be used in different ways:

- if the logical structure (XML) of the document is considered in isolation, XML can be regarded as being the format for information exchange between users, applications, etc;
- if we take into account the physical layout, we obtain:
  - an independent publication format to show said structured content to the user;
  - if we made use of the stylesheet, we could modify the presentation of the document to personalize the information. The same XML document and different stylesheets can generate different versions for different users. perhaps with more or less content as appropriate for different information spaces (on-line version, printed version, etc.);
- if links are added we get a hyperdocument which relates elements of different documents, allowing one to "navigate" through the information.
- if code (java, javascript, etc.) is added to the document, the document will become an application. It will be able to interact with the user (person, intelligent agent or application) and respond to events.

  To process an XML document there exist two formats, object oriented processing (eg. DOM [5]) and event oriented processing (eg. SAX), which will be described with more detail in the following section.

1.2 What does XML look like?

The appearance of an XML document is very similar to that of SGML or HTML documents, as can been seen from figure 2:



```
<html>
 <dl>
  <dt>Hot Cop
   <dd>by Jacques Morali,
          Henry Beolo, and Victor Wills
     <ul>
       <li>producer: Jacques Morali </li>
       <li>publisher: PolyGramRecords</li>
       <li>length: 6:20</li>
       <li>written: 1978 </li>
       <li>artist: Village People<li>
     </ul>
</dl>
</html>
```

```
<xml>
 <song>
   <title>Hot Cop</title>
   <composer> Jacques Morali</composer>
   <composer> Henry Beolo </composer>
   <composer> Victor Wills </composer>
   <producer> Jacques Morali </producer>
   <publisher> PolyGram </publisher>
   <lenght> 6:20 </length>
   <year> 1978 </year>
   <artist> Village People </artist>
 </song>
</xml>
```

```
Hot Cop
 by Jacques Morali,   Henry Beolo, and Victor Wills
     •producer: Jacques Morali
     •publisher: PolyGramRecords
     •length: 6:20
     •written: 1978
     •artist: Village People
```

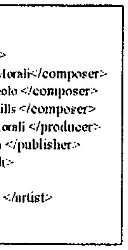


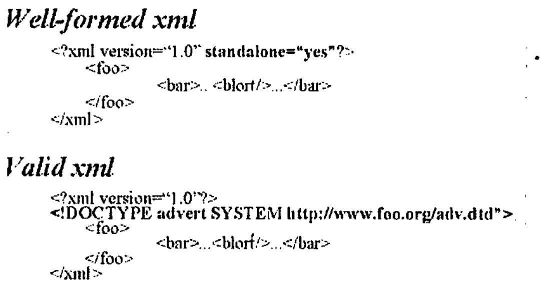
Figure 2: XML vs HTML

XML documents are comprised of marker and content tags. The five marker tags are the following:

- elements
- entity references
- comments
- processing instructions
- sections CDATA

Types of XML documents Those documents that comply with the basic rules of the syntax of XML are said to be well formed documents. Those that depart from the rules must follow a structure defined by the user and specified via a DTD document. When an XML document complies with the DTD, it is said to be a valid document. Parsers must verify the validity of the XML document with respect to the DTD.

As can be seen in figure 3, the header of the document specifies whether it is to be a well formed XML document or valid with respect to some DTD.



Well-formed xml

```
<?xml version="1.0" standalone="yes"?>
    <foo>
              <bar>.. <blort/>...</bar>
    </foo>
<//xml>
```

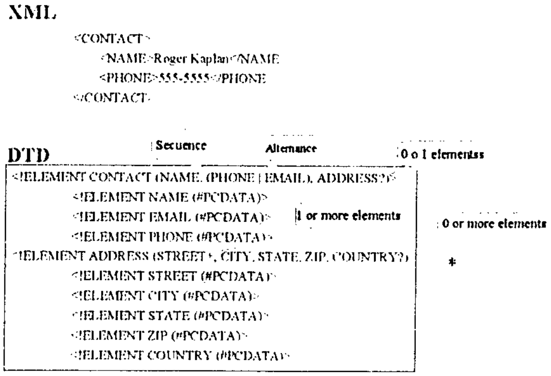
Valid xml

```
<?xml version="1.0"?>
<!DOCTYPE advert SYSTEM http://www.foo.org/adv.dtd">
    <foo>
              <bar>...<blort/>...</bar>
    </foo>
</xml>
```

Figure 3: XML header

Predefined structures There are two forms of defining the structure of an XML document: using a DTD, or using XML Schemas.

The DTD is a concept inherited from SGML. A DTD allows us to be sure that an XML document complies with a predetermined structure, but it brings with it a few problems: it has a different syntax to XML and it does not permit the definition of new "basic data types" (int, boolean, etc.). From this arose the necessity of developing a new standard to describe the structure of a XML document: XML-Schemas [7].



XML

```
<CONTACT>
    <NAME>Roger Kaplan</NAME>
    <PHONE>555-5555</PHONE>
</CONTACT>
```

DTD

```
<!ELEMENT CONTACT (NAME, (PHONE | EMAIL), ADDRESS?)>
        <!ELEMENT NAME (#PCDATA)>
        <!ELEMENT EMAIL (#PCDATA)>
        <!ELEMENT PHONE (#PCDATA)>
<!ELEMENT ADDRESS (STREET+, CITY, STATE, ZIP, COUNTRY?)
        <!ELEMENT STREET (#PCDATA)>
        <!ELEMENT CITY (#PCDATA)>
        <!ELEMENT STATE (#PCDATA)>
        <!ELEMENT ZIP (#PCDATA)>
        <!ELEMENT COUNTRY (#PCDATA)>
```

Secuence
Alternance
0 o 1 elementss
1 or more elements
0 or more elements
*

Figure 4: XML DTD

XML Schemas Schemas permit the definition of the structure of a document using the syntax of XML. This makes the structure more legible. Schemas also permit the definition of new data types (String, integer, etc.).

Which one to use, DTDs or Schemas? In general, in applications orientated towards documents where there are no strong typing implicit in the data, it is better to use DTDs. In those applications where data typing is important, Schemas are best used.

1.3 A set of types for XML

In giving XML a type system, first a set of formal types for XML documents must be given, and then the relations between them, i.e. a *semantics* of types for XML, must be stated. We will want at least the following semantic equations to hold in accordance with the perception that the XML documents they describe are not different on paper. Respectively: grouping within a sequence of elements does not matter; a list type following its base type is the same as the base type following the sequence type – both describe lists of length at least one; a list of lists is the same as a list type. I.e. there is no recording of how a sequence has historically been constructed.

$$
\begin{array}{rcll}
(x,(y,z)) & = & ((x,y),z) & = (x,y,z) \\
(x,x^*) & = & (x^*,x) & \leq x^* \\
x^{**} & = & x^* &
\end{array}
$$

We want documents satisfying XML schema containing only variants such as the above to be typed equally. It should be clear that the *sequence* of types in the interior of a tagged structure is the item of importance. The complete list of possible formal types that we will consider is given in (3) below:

$$
\begin{array}{rcll}
\text{Type} & ::= & a\,t & \text{tagged structure} \\
 & | & (t_1,\ldots) & \text{products, found between tags} \\
 & | & t_1 + t_2 & \text{unions} \\
 & | & t^* & \text{unbounded finite sequences} \\
 & | & \epsilon & \text{empty type} \\
 & | & \omega & \text{all type} \\
 & | & \mu x.\, t & \text{fixpoint solution to } x = t[x] \\
 & | & \texttt{String} & \text{strings} \\
 & | & v & \text{type variable}
\end{array}
\tag{3}
$$

where $t, t_1, t_2 \in$ Type and a is a tag identifier. Even if they are semantically equal, we want to be able to *write* $(a,(b,c))$ and $((a,b),c)$ and (a,b,c) differently. In the next section we will begin to dress this data type up with some, initially, formal semantics.

1.4 A wish-list for reasoning about types

In this section we project some rules for reasoning about types in a formal way, before going on in the next section to provide a semantic substrate in which these rules may be derived, rather than simply formulated. The rules express intuitions about XML types.

Clearly, yet more type equations and inequations will follow from the most basic intuitions about XML types expressed in the opening section, such as:

$$\begin{aligned}(x+y,z) &= (x,y)+(y,z)\\ (x,y+z) &= (x,y)+(x,z)\\ (x+y)* &\geq x**+y*\end{aligned}$$

What is needed to make these equations hold is the standard set-theoretic interpretation of types, but with the nuance that type products are more like set-theoretic sequences than set-theoretic products. x^* must be the limit of the operation of extending a product type through (x) and (x,x) and (x,x,x) ..., and including all the different length products as subtypes, so that

$$(x,x*) = (x,\epsilon+(x,\epsilon+(x,\ldots)) = (x)+(x,x)+(x,x,x)+\ldots \leq x^*$$

This intention can be expressed via the rules immediately below:

$$\text{(list intro.)}\quad \frac{}{(a,a^*)\leq a^* \quad (a^*,a)\leq a^* \quad \epsilon\leq a^* \quad a\leq a^*} \tag{4}$$

$$\text{(list elim.)}\quad \frac{\epsilon\leq b \quad \forall x.\ x\leq b \rightarrow (a,x)\leq b \quad a^*\leq b\rightarrow p}{p} \tag{5}$$

$$\text{(list elim.)}\quad \frac{\epsilon\leq b \quad \forall x.\ x\leq b \rightarrow (x,a)\leq b \quad a^*\leq b\rightarrow p}{p} \tag{6}$$

The rules above are presented in a somewhat unnatural form in order that they fit the pattern of *introductions* – in which the constructs below the line are more complex than those above – and *eliminations* – in which the converse is true. But they are recasts of more familiar reasoning procedures.

The list introduction rules above assert that the list type contains the closure of the sets obtained by continuously extending a product by the same type again and again. The list elimination rules state that it is the least such closure. That is to say that the type is a limit type, the solution to a type equation:

$$\begin{aligned}a^* &= \epsilon+(a,a^*)+(a^*,a)\\ &= \mu x.\ \epsilon+(a,x)+(x,a)\end{aligned}$$

but the list type deserves a name of its own! Written in a more recognizable form, list elimination says, replacing p by the formula $a^*\leq b$:

$$\frac{\epsilon\leq b \quad \forall x.\ x\leq b\rightarrow (x,a)\leq b}{a^*\leq b}$$

which is evidently the statement that the list type is the minimal of the closures of the empty type under postfixing.

To the rules for list types, we have to add the ordinary set-theoretic rules for inclusion, unions and products:

$$\text{(refl.)}\quad \frac{}{x\leq x} \tag{7}$$

$$\text{(trans.)}\quad \frac{x\leq y \quad y\leq z}{x\leq z} \tag{8}$$

$$\text{(all intro.)}\quad \frac{}{x\leq \omega} \tag{9}$$

$$\text{(all elim.)}\quad \frac{\forall y.\ y\leq x \quad \omega\leq x\rightarrow p}{p} \tag{10}$$

$$\text{(union intro.)}\quad \frac{x \le a}{x \le a+b} \quad \frac{x \le b}{x \le a+b} \tag{11}$$

$$\text{(union elim.)}\quad \frac{a \le x \quad b \le x \quad a+b \le x \to p}{p} \tag{12}$$

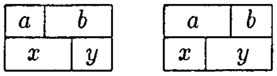
$$\text{(prod. intro.)}\quad \frac{x \le a \quad y \le b}{(x,y) \le (a,b)} \tag{13}$$

$$\text{(prod. elim.)}\quad \frac{(x,y) \le (a,b) \to p \quad (x,t) \le a \wedge y \le (t,b) \vee x \le (a,t) \wedge (t,y) \le b}{p} \tag{14}$$

These rules identify inequality between types as essentially set-theoretic inclusion, with a top class ω. Union is the minimal set containing both components, and products are really set-theoretic senquences, with their elements "promoted" so that if they themselves contain sequences, then the elements of the elements appear inline at top-level within the sequence. The rule given for product elimination can be understood more easily by replacing p with $(x,y) \le (a,b)$. Then the rule reads:

$$\frac{(x,t) \le a \wedge y \le (t,b) \vee x \le (a,t) \wedge (t,y) \le b}{(x,y) \le (a,b)}$$

which is the law given by Tarski in his treatises on ordinal algebras and which describes the interaction between inclusion and sum on total orders. Here it has validity because it describes how two partitions $(a.b)$ and (x,y) of the same sequence of elements may be related. Either a is short and has to be padded by some t to be the same length as x, or a is long and it is x that has to be padded by t to reach the length of a. In the first case we can compare (short) a followed by t with (long) x, and t followed by (short) y with (long) b. In the second case, (long) a can be compared with (short) x followed by t, and t followed by (short) b with y. The two cases are depicted below:



The empty type is the identity for type products. When it occurs in a sequence it can be forgoten. In the trivial case, a product of one type is just the same as a type alone:

$$\text{(empty intro.)}\quad \frac{x \le y}{(\epsilon,x) \le y \quad (x,\epsilon) \le y \quad x \le (\epsilon,y) \quad x \le (y,\epsilon)} \tag{15}$$

$$\text{(empty elim.)}\quad \frac{(\epsilon,x) \le y}{x \le y} \quad \frac{x \le (\epsilon,y)}{x \le y} \quad \frac{(x,\epsilon) \le y}{x \le y} \quad \frac{x \le (y,\epsilon)}{x \le y} \tag{16}$$

$$\text{(singleton intro.)}\quad \frac{x \le y}{(x) \le y \quad x \le (y)} \tag{17}$$

$$\text{(singleton elim.)}\quad \frac{x \le (y)}{x \le y} \quad \frac{(x) \le y}{x \le y} \tag{18}$$

The rules governing the fixpoint operator identify it as finding the fixpoint solution, and being least with that property:

$$\text{(fix intro.)}\quad \frac{}{e[(\mu x.e)/x] \le \mu x.e} \tag{19}$$

$$\text{(fix elim.)}\quad \frac{\forall x.\ x \le t \to e \le t \quad \mu x.\ e \le t \to p}{p} \tag{20}$$

and, finally, the tag type constructor does nothing to its elements except "tag" them, so it preserves and reflects inequalities:

$$\text{(tag intro.)}\quad \frac{x \le y}{a\ x \le a\ y} \tag{21}$$

$$\text{(tag elim.)}\quad \frac{a\ x \le a\ y}{x \le y} \tag{22}$$

Though these rules presently compose nothing more than a *wish-list*, they are *sound*. This is because they really do have a *model*. That model interprets the types as *parsers*, and is given in the next section. This is the essence of the insight that XML documents can be typed, for types in general might be anything at all that proves convenient as an *a priori* calculation in order to help ensure the syntactic correctness of code, but the distinguishing feature of XML is that it is a language that treats of documents, so the types of XML must be calculations that check the correctness of documents, i.e. parsers.

The rules given have also been carefully formulated in terms of introductions – which construct a relation with more complicated expressions within – and eliminations – which reduce complexity. The idea in setting them up in this way is that it may then be shown that an introduction followed by an elimination can be recast as an elimination followed by an introduction. That means that proofs using these rules can be reformulated as a sequence of eliminations followed by introductions along any descending branch of the proof tree. They must then pass through an assertion about the relation between certain subexpressions of the original along the way. Further, uses of transitivity and reflexivity (for non-atomic types) can be removed from the proof. These observations mean that proofs of an inclusion between given types are essentially *normalizable*, and render searches for proofs – in principle – mechanizable.

2 The semantics of XML types

In the last section, we concluded that *XML types are parsers*. This is another way of saying that from each type, we can construct a parser for the documents described by the type. Viewing that yet another way, XML schemas and document descriptions can be seen as the instructions how to build a type-directed parser. That is, each schema tells how to construct a parser for its documents. The parser construction depends only on the type expressed by the schema, and no more information. In this section details of the construction are given.

The constructed parsers are context-free ambiguous parsers. From each XML document they produce a set of alternative parses. Each parse is a "syntax tree", or more exactly the Tree type described in equations (1) and (2), but in general it could be anything, so the syntax tree type appears as a variable in the definition (23) below:

$$\text{Parser } x \;=\; \text{Text} \to \{\, x \,\} \tag{23}$$

In the following section, the semantics of these parsers will be explored, and then the type constructors of XML will be expressed in terms of operations on parsers.

2.1 The algebra of parsers

The ambiguous parsers defined in (23) admit an algebra which includes the principal operations of placing parsers in sequence and in alternation. Respectively:

$$\begin{aligned}
(\frown) \;&::\; \text{Parser } [x] \to \text{Parser } [x] \to \text{Parser } [x] \\
(p_1 \frown p_2)\mathit{text} \;&=\; \{s_1 \frown s_2 | x_1 \frown x_2 = \mathit{text},\ s_1 \in p_1\ x_1,\ s_2 \in p_2\ x_2\} && (24)\\
(+) \;&::\; \text{Parser } x \to \text{Parser } x \to \text{Parser } x \\
(p_1 + p_2)\mathit{text} \;&=\; p_1\ \mathit{text} \cup p_2\ \mathit{text} && (25)
\end{aligned}$$

That is, sequencialization entails the partition of the text in every way possible, and the concatenation of the sequences of trees that result from the analysis of each set of two parts. The ambiguous nature of the parsers ensures a large number of possibilities. Alternation is the set-theoretic union of the possible parses from either one of the parsers.

As is to be expected, sequentialization distributes over alternation. That is:

$$\begin{aligned}
a \frown (b + c) \;&=\; (a \frown b) + (a \frown c) \\
(a + b) \frown c \;&=\; (a \frown c) + (b \frown c)
\end{aligned}$$

The sequencialization operator, which operates between parsers that produce sequences of syntax trees, can be expressed in terms of more elementary operators. If ϵ is the parser which recognises the empty text (and produces the empty sequence of syntax trees), and : is the operation between parsers described immediately below, then:

$$(p_1 : \epsilon) \frown p_2 \;=\; p_1 : p_2$$

The combinators ϵ and : have precisely the following semantics, analogous respectively to that of the empty list and the 'cons' operator which adds a new element to the front of a list:

$$\begin{aligned}
(:) \quad &:: \quad \text{Parser } x \rightarrow \text{Parser } [x] \rightarrow \text{Parser } [x] \\
(p_1 \;:\; p_2)\mathit{text} \quad &= \quad \{r_1 : s_2 \mid x_1 \frown x_2 = \mathit{text},\ r_1 \in p_1\ x_1,\ s_2 \in p_2\ x_2\} \qquad (26) \\
\epsilon \quad &:: \quad \text{Parser } [x] \\
\epsilon \ulcorner\,\urcorner \quad &= \quad \{\,[\,]\,\} \qquad (27) \\
\epsilon\, \_\_ \quad &= \quad \{\,\}
\end{aligned}$$

The ϵ parser recognizes precisely the empty text. It rejects any other text. That is in contrast to the the error parser, which rejects every text, and the string parser, which accepts precisely a text composed of one string, and produces the leaf syntax tree that corresponds to it:

$$\begin{aligned}
\text{error} \quad &:: \quad \text{Parser } x \\
\text{error}\, \_\_ \quad &= \quad \{\,\} \qquad (28) \\
\text{string} \quad &:: \quad \text{Parser Tree} \\
\text{string} \ulcorner \mathit{str} \urcorner \quad &= \quad \{\ \mathit{str}\ \} \qquad (29) \\
\text{string}\, \_\_ \quad &= \quad \{\ \ \}
\end{aligned}$$

The ϵ parser is the identity for sequencialization, whereas the error parser is the identity for alternation:

$$\begin{aligned}
a \frown \epsilon \quad &= \quad \epsilon \frown a \quad = \quad a \\
a + \text{error} \quad &= \quad \text{error} + a \quad = \quad a
\end{aligned}$$

The combination $p_1 \;:\; \epsilon$ performs a trivial change on the trees produced by the parser p_1: it inserts each in a singleton list. The operation changes the type but not the essence of the data, and it is therefore of utility when combining parsers, because it "promotes" the syntax trees that they produce up by one list bracket.

$$\begin{aligned}
\text{singleton} \quad &:: \quad \text{Parser } x \rightarrow \text{Parser } [x] \\
\text{singleton } p \quad &= \quad p : \epsilon \qquad (30)
\end{aligned}$$

The converse operation is also useful. The flatten operator removes a pair of list brackets, converting a parser that produces a sequence of sequences of syntax trees into one that produces a sequence of syntax trees.

$$\begin{aligned}
\text{flatten} \quad &:: \quad \text{Parser } [[x]] \rightarrow \text{Parser } [x] \\
\text{flatten } p\ \mathit{text} \quad &= \quad \{\ \text{concat } \mathit{ss} \mid \mathit{ss} \in p\ \mathit{text}\ \} \qquad (31)
\end{aligned}$$

Finally, we can construct a parser that accepts an arbitary number of repeats of a given document design:

$$\begin{aligned}
(*) \quad &:: \quad \text{Parser } x \rightarrow \text{Parser } [x] \\
p^* \quad &= \quad x \text{ where } x = \epsilon + (p : x) \qquad (32)
\end{aligned}$$

In the next section, the operations detailed above serve to express the semantics of XML types. Obviously, given their setting as multi-valued functions, parsers have limits of increasing sequences, where one parser is said to be greater than the other if it is more "accepting". That is, all the parses of the smaller parser are also parses of the larger parser:

$$\begin{aligned}
p_1 \leq p_2 \quad &\leftrightarrow \quad \forall \mathit{text}.\ p_1\ \mathit{text} \subseteq p_2\ \mathit{text} \qquad (33) \\
&\leftrightarrow \quad p_1 + p_2 = p_2 \qquad (34)
\end{aligned}$$

The limit is the set theoretic limit:

$$p_1 \leq p_t \leq \ldots \nearrow p \;\leftrightarrow \forall \mathit{text}.\; p\; \mathit{text} \;=\; \bigcup_i p_i\; \mathit{text} \tag{35}$$

and all the (finitary) operators already discussed are monotonic with respect to the ordering (34) on parsers, and continuous with respect to directed limits.

2.2 XML types as parsers

The construction of parsers from XML types is "context free", in the sense that it depends only on the type expression, not on the context in which the type expression is embedded. First of all, given a tag type, the corresponding parser checks that the tag is what it should be, and then parses the content inside the tag. The result will be a set of possible singleton lists of parse trees. The singleton represents the unitary character of a tag, as compared to the multiple character of the sequence of contents inside the tag:

$$\begin{array}{rcl} [\![-]\!]_C & :: & \text{Type} \to \text{Parser [Tree]} \\ [\![a\ t]\!]_C \ulcorner\texttt{<}a\texttt{>}\ \mathit{text}\ \texttt{</}a\texttt{>}\urcorner & = & \{\ [\langle a\rangle\ x\ \langle /a\rangle] \mid x \in [\![t]\!]_C\ \mathit{text}\ \} \end{array} \tag{36}$$

where e is a type expression that commences with a tag construct.

The interpretation of the content type within a tag is a parser that produces a nontrivial list of trees, not a singleton. Each tree in the list corresponds to a (promoted) element in the tag contents.

For example, the simple tagged document:

```
<a> "Hello" "World" </a>
```

satisfies the type

$$\text{a}\ (\texttt{String}, \texttt{String})$$

and the parser for this type, when applied to the document, will produce the unique parse tree with a single node and two leaf branches from it:

$$\langle \text{a}\rangle\ [\texttt{"Hello"}, \texttt{"World"}]\ \langle /\text{a}\rangle$$

so the parser for the product type within the tag produces lists of trees, which can then be fixed as branches to the tag node above.

$$\begin{array}{rcll} [\![(t_1,\ldots,t_n)]\!]_C & = & [\![t_1]\!]_C \frown \ldots \frown [\![t_n]\!]_C & (37) \\ [\![(\,)]\!]_C & = & \text{empty} & (38) \\ [\![t^*]\!]_C & = & \text{flatten } ([\![t]\!]_C)^* & (39) \\ [\![t_1 + t_2]\!]_C & = & [\![t_1]\!]_C + [\![t_2]\!]_C & (40) \\ [\![\texttt{String}]\!]_C & = & \text{singleton string} & (41) \\ [\![\mu x.\, e]\!]_C & = & [\![\theta]\!]_C \text{ where } \theta = e[\theta/x] & (42) \\ [\![\epsilon]\!]_C & = & \text{empty} & (43) \end{array}$$

The fixpoint interpretation deserves a little comment. It is the interpretation of the fixpoint of a type equation, which itself will yield an infinitely extended formal type expression (without the μ operator) provided there is at least a leading constructor on the type. So $\mu x.a\ x$ for some tag a is alright, but $\mu x.x$ is not. The interpretation of the infinitely extended type expression goes as follows: the rules above tell one how to interpret the approximants to the type as parsers, and the interpretation is the functional limit of those interpretations.

It is also possible to define the interpretation of $\mu x.e$ to be the fixpoint of the automorphism of parsers $p \mapsto [\![e]\!]'_C$ in which the interpretation $[\![-]\!]'_C$ now transforms the type variable x to p when it encounters it in expression e. That calculation should give the identical result because the extra information fed into the functional fixpoint calculation is that the interpretation of x will be a parser, which is already inevitable because it must fit into the operators that are applied to it in the interpreted expression.

It is possible to define the list type as a limit of types:

$$t^* = \mu x.\ (\epsilon + (t, x))$$

and this will yield the same parser as the list type parser described by the earlier equations.

For example, if we try the parsers obtained, respectively, from the types

$$\begin{array}{l} \mathrm{a}\ \mathtt{String}^* \\ \mathrm{a}\ (\mathtt{String}, \mathtt{String}^*) \\ \mathrm{a}\ (\mathtt{String}^*, \mathtt{String}) \\ \mathrm{a}\ (\mathtt{String}^*, (\mathtt{String})) \\ \mathrm{a}\ (\mu x.\epsilon + (\mathtt{String}, x), (\mathtt{String})) \end{array}$$

on the document

```
<a> "Hello World" </a>
```

(a tag pair containing a string) then all the parsers parse the document in exactly the same way, as:

$$\{\ \langle \mathrm{a} \rangle\ \texttt{"Hello World"}\ \langle /\mathrm{a} \rangle\ \}$$

(the outfix constructor enclosing a leaf node). The parse is a singleton, indicating that there are no alternative parses of the document. However, the parsers are not exactly the same, because the first admits the empty tag:

$$\langle \mathrm{a} \rangle\ \langle /\mathrm{a} \rangle$$

and the rest do not. But the other parsers are identical, and all parse everything in the same way, with the single exception noted. Therefore the other parsers are subparsers of the first and the type correspondence is:

$$\begin{array}{rcccl} \mathrm{a}\ \mathtt{String}^* & \geq & \mathrm{a}\ (\mathtt{String}, \mathtt{String}^*) & = & \mathrm{a}\ (\mathtt{String}^*, \mathtt{String}) \\ & = & \mathrm{a}\ (\mathtt{String}^*, (\mathtt{String})) & = & \mathrm{a}\ (\mu x.\epsilon + (\mathtt{String}, x), (\mathtt{String})) \end{array}$$

The rules for the comparison of types can be derived from the algebra and semantics of operations on parsers.

The parser constructed from a type parses the text iff text conforms to the type. That intention is expressed formally as follows:

$$[\![t]\!]_R\ \text{text} \neq \{\ \} \leftrightarrow t\ \models\ \text{text}$$

But it is practically difficult to verify this intention by any other means than parsing using the parser constructed from the type. Some texts obviously conform to the specification given for them: they may, for example, consist of a sequence of strings within a single tag. The document type specification should then say "sequence of strings within a single tag". The problem arises when the document specification says something more complicated, such as "sequence of sequences of strings all within a single tag". Then the type is not self-evidently that which it should be. In this example, the type is *equal* to the evident type of the document, but not *identical*. The variants among type expressions that are equal but not identical may in principle be so great as to leave no obvious alternative as to try the parser constructed from the type on the document to see if it conforms. There will be many different alternative parses to try, since the parsers are ambiguous.

In the next section, an algorithm is presented which is successful in identifying many of the possible variants of type presentations.

2.3 An algorithm for the comparison of types

The reduction rules below do a remarkably good job of assigning a relation of "formal inclusion" for types. The open end of the relation symbol points to the larger type, which is the more accepting parser:

$$t_1 + t_2 \preceq x \leftarrow t_1 \preceq x \wedge t_2 \preceq x \tag{44}$$

$$x \preceq t_1 + t_2 \leftarrow x \preceq t_1 \vee x \preceq t_2 \tag{45}$$

$$\texttt{String} \preceq \texttt{String} \tag{46}$$

$$x^* \preceq y^* \leftarrow x \preceq y \tag{47}$$

$$(t_1, \ldots, t_n) \preceq x^* \leftarrow t_1 \preceq x^* \wedge \ldots t_n \preceq x^* \tag{48}$$

$$x \preceq y^* \leftarrow x \preceq \epsilon \vee x \preceq y \tag{49}$$

$$x \preceq (t_1, \ldots, t_k, (t_{k+1}, \ldots, t_l), t_{l+1}, \ldots, t_n) \leftarrow x \preceq (t_1, \ldots, t_n) \tag{50}$$

$$(t_1, \ldots, t_k, (t_{k+1}, \ldots, t_l), t_{l+1}, \ldots, t_n) \preceq y \leftarrow (t_1, \ldots, t_n) \preceq y \tag{51}$$

$$x \preceq (t_1, \ldots, t_k, \epsilon, t_{k+1}, \ldots, t_n) \leftarrow x \preceq (t_1, \ldots, t_n) \tag{52}$$

$$(t_1, \ldots, t_k, \epsilon, t_{k+1}, \ldots, t_n) \preceq y \leftarrow (t_1, \ldots, t_n) \preceq y \tag{53}$$

$$(t_1, \ldots, t_n) \preceq (t'_1, \ldots, t'_n) \leftarrow t_1 \preceq t'_1 \wedge \ldots \wedge t_n \preceq t'_n \tag{54}$$

$$(x) \preceq y \leftarrow x \preceq y \tag{55}$$

$$x \preceq (y) \leftarrow x \preceq y \tag{56}$$

$$a\ t_1 \preceq a\ t_2 \leftarrow t_1 \preceq t_2 \tag{57}$$

$$x \preceq (\,) \leftarrow x \preceq \epsilon \tag{58}$$

$$(\,) \preceq y \leftarrow \epsilon \preceq y \tag{59}$$

$$\epsilon \preceq \epsilon \tag{60}$$

$$x \preceq \omega \tag{61}$$

$$\mu x.\ e_1 \preceq \mu x.\ e_2 \leftarrow e_1 \preceq e_2 \tag{62}$$

with equality of type variables of the same name within expressions. The error parser is the smallest parser, but no type translates to it. Note that reflexivity ($x \preceq x$) is not expressed generically, but only of atomic types. The recursive application of the rules reduces the assertion of reflexivity to reflexivity on atomic types.

Note that the rule applications always terminate because the type expressions on the right hand side of each rule are always simpler than those on the left hand side.

In principle this is a *proper* subrelation of the functional definition of when one parser is more accepting than another. That is:

$$t_1 \preceq t_2 \rightarrow t_1 \leq t_2 \tag{63}$$

yet it is a larger subrelation than might naively be supposed, because, for example, it correctly determines the inclusion between a list type and the type of lists of at least one entry:

$$(\texttt{String}, \texttt{String}^*) \preceq \texttt{String}^*$$

Where the subrelation is incomplete is in the comparison of types that can be proved to be equal via set-theoretic reasoning, but which are not self-evidently so from the syntax alone. The list type cannot be determined to be equal to its definition as a set-theoretic limit, for example:

$$\mu x.\epsilon + (\texttt{String}, x) \not\preceq \texttt{String}^*$$

However, another pair of reduction rules may be added to take care of the comparisons between lists and limits:

$$x^* \preceq \mu v.e \leftarrow \epsilon + (x, v) \preceq e \tag{64}$$

$$\mu v.e \preceq y^* \leftarrow e \preceq \epsilon + (y, v) \tag{65}$$

That leaves the problem that different partitions of a document are not detected by these rules. For example:

$$(\mathtt{String}, (\mathtt{String}, \mathtt{String})) \not\preceq ((\mathtt{String}, \mathtt{String}), \mathtt{String})$$

That is because there is no convenient way to detect all the Tarskian relationships:

$$(a,b) \preceq (c,d) \leftarrow \exists x.\ (a,x) \preceq c \wedge b \preceq (x,d) \vee a \preceq (c,x) \wedge (x,b) \preceq d \tag{66}$$

but substituting for x in the above in turn all the possible subexpressions of a, b, c, d appears to be sufficient for the common cases. In particular, adding the rule (66) above with x restricted to the proper subexpressions of the left and right hand sides does make the mentioned inequality hold:

$$(\mathtt{String}, (\mathtt{String}, \mathtt{String})) \preceq ((\mathtt{String}, \mathtt{String}), \mathtt{String})$$

The search can be lengthy in the case that the result is false, because the set of subexpressions might be recalculated for the recursive comparisons of every subexpression. Considerable efficiency can be achieved by calculating the set once for the whole, at the outset, and then reusing it throughout the recursion. Even the removal of repeats from the list of subexpressions may incurr a comparison for equality, which will evoke another recursive calculation, so there is much opportunity to produce a very computationally expensive implementation of the reduction rule (66).

Similarly, because the list type corresponds to the union of various sequence types, an extra rule must be inserted to cope with the comparison $(a,b) \leq t^*$. It is derived from (66) by substituting (c,d) by (t,t^*). A symmetrically derived variant replaces (c,d) by (t^*,t):

$$(a,b) \preceq t^* \quad \leftarrow \quad \exists x.\ (a,x) \preceq t \wedge b \preceq (x,t^*) \vee a \preceq (t,x) \wedge (x,b) \preceq t^* \tag{67}$$
$$(a,b) \preceq t^* \quad \leftarrow \quad \exists x.\ (a,x) \preceq t^* \wedge b \preceq (x,t) \vee a \preceq (t^*,x) \wedge (x,b) \preceq t \tag{68}$$

Obviously $(a,b) \leq (t,t,t^*)$ etc., etc. remain to be considered too. But replacing the fragment x by (t,x) in the above formulae does the job. The trouble is that x is supposed to be a subexpression of those that appear in the original formula, and that is not necessarily the case for (t,x). One must accept the incompleteness of this search procedure or be prepared to try comparisons of unbounded length. In practice, searching to a depth of two or three repeats will be enough.

We are still not finished, because the rules above are in principle not enough to resolve when to insert extra empty types in a sequence for the purposes of comparision. That is:

$$(\mathtt{String}) \not\preceq (\mathtt{String}, \mathtt{String}^*)$$

because although the extra relations

$$(\mathtt{String}) \preceq (\mathtt{String}, \epsilon),\ (\mathtt{String}, \epsilon) \preceq (\mathtt{String}, \mathtt{String}^*)$$

are known, there is no rule that creates these intermediate results and then applies transitivity to them. That can be remedied with the rule below, which checks to see if some elements of a sequence type contain the empty type, and thus may be discounted in the comparison:

$$x \preceq (a,b) \quad \leftarrow \quad x \preceq a \wedge \epsilon \preceq b \vee \epsilon \preceq a \wedge x \preceq b \tag{69}$$

With this rule in operation, the example is resolved correctly:

$$(\mathtt{String}) \preceq (\mathtt{String}, \mathtt{String}^*)$$

and indeed even other nonobvious relations are calculated correctly by the rules. For example:

$$\begin{array}{rcl} (a,a,a) & \preceq & (a+b)^* \\ (a,b,a,b,a) & \preceq & ((a,b)^*,a) \end{array}$$

Are there still relations that cannot be uncovered by these rules? Yes, there are. As remarked, those which derive from the unrolling of limits (such as list types) to some a priori unknown depth before they become true.

3 Summary

In this article we have presented XML and a system of types for it, interpreting the types as parsers in order to get the required semantics for the inclusion relations between types. Obviously, the membership of a document in a type is decidable – the decision is made by parsing the document using the parser corresponding to the type. Although the full relation of inclusion between types appears not to be decidable, most practical cases can be rendered via the algorithm presented here.

References

[1] Philip Wadler. *The next 700 Markup Languages*, Second Conference on Domain Specific Languages (DSL'99), Austin, Texas, October 1999.

[2] A. Brown , M. Fuchs, Jonathan Robie and Philip Wadler. *A model for W3C XML Schema.* WWW10, Hong Kong, May 2001.

[3] E. Hehner. *A practical theory of programming*, Springer Verlag, 1993.

[4] XPointer. WD 6-Dec-99. `http://www.w3.org/ TR/xptr`.

[5] DOM. CR 10-May-00. `http://www.w3.org/ TR/DOM-Level-2`.

[6] XLink. WD 21-Feb-00. `http://www.w3.org/TR/ xlink`.

[7] XML Schemas . `http://www.w3.org/TR/xmlschema-0/` `http://www.w3.org/TR/xmlschema-1/` `http://www.w3.org/TR/xmlschema-2/`.

[8] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, `http://w3c.org/ TR/REC-xml`, W3C, 10th February, 1998.

[9] XSL. WD 27-Mar-00. `http://www.w3.org/ TR/xsl/`.

[10] XSLT 1.0. R 16-Nov-99. `http://www.w3.org/ TR/xslt`.

Automatic Test Generation from Specifications for Control-Flow and Data-Flow Coverage Criteria

Hyoung Seok Hong and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
{hshong@saul., lee@}cis.upenn.edu

Abstract

This paper presents an approach to automatic test generation from specifications written in extended finite state machines (EFSMs). We consider a hierarchy of coverage criteria based on the information of control flow and data flow in EFSMs and formulate the problem of test generation as finding counterexamples during the model checking of EFSMs. The ability of model checkers to construct counterexamples allows test generation to be automatic.

We illustrate our approach using the temporal logic CTL and its symbolic model checker SMV. We show how to translate an EFSM into a SMV program and how to express a coverage criterion as a set of CTL formulas. A test suite consists of all the counterexamples obtained by model checking the CTL formulas against the SMV program.

Keywords: test generation, extended finite state machines, control flow and data flow, model checking

1 Introduction

Testing has always been an essential activity for validating the correctness of software and hardware systems. The testing process usually consists of two main steps: test generation and test execution. Test generation constructs tests from the specification of an implementation under test by determining which functionalities will be tested. Test execution applies tests to the implementation, makes observations during the execution of tests, and validates the observed behaviors. Although testing cannot provide an absolute guarantee on correctness as is possible with mathematical verification, systematic test generation and execution can greatly increase the effectiveness of system validation, especially when performed automatically by suitable tools.

This paper discusses the problem of test generation from formal specifications written in extended finite state machines (EFSMs). EFSMs extend FSMs with data variables and operations on them and are widely used as underlying models of many specification languages such as SDL [2], Estelle [4], and Statecharts [11]. Because an EFSM specification typically allows an infinite number of executions, it is impossible to determine whether an implementation conforms to its specification by considering all the possible executions of the specification. Therefore, exhaustive testing is in general impossible to achieve and it is necessary to have systematic coverage criteria which select a finite and reasonable number of tests satisfying certain conditions. In the last two decades, a number of test generation methods and tools have been proposed for EFSMs (for survey, see [3, 8, 17]) and most of them focus on a hierarchy of coverage criteria based on the information of control flow (e.g., states and transitions) or data flow (e.g., definitions and uses of variables) in EFSMs.

This paper shows that the problem of test generation from EFSMs based on control and data flow coverage criteria can be formulated as a model checking problem. Given a finite state system model and a temporal logic formula, model checking provides either a claim that the formula is satisfied in the system model or else a counterexample explaining why the formula is falsified. In our approach, each coverage criterion is expressed as a set of temporal logic formulas and the problem of test generation satisfying the criterion is formulated as finding counterexamples during model checking the formulas against EFSMs. The ability of model checkers to construct counterexamples allows test generation to be automatic.

We illustrate our approach using the temporal logic CTL [7] and its symbolic model checker SMV [18]. An overview of test generation is shown in Figure 1. First we describe how to translate EFSMs into input to SMV, which we call SMV program. We, then, describe how to express coverage criteria in CTL. A given coverage criteria is expressed as a parameterized set of formulas in CTL that are instantiated for a given EFSM specification. Each formula is defined such that the formula is satisfied in an EFSM if and only if the EFSM does *not* have an execution which covers the entity described by the formula. If the entity can be covered in the EFSM, model checking will fail and SMV will generate a counterexample which is one of the executions covering the entity. A test suite consists of all the counterexamples obtained by model checking each CTL formula in the formula set against the SMV program.

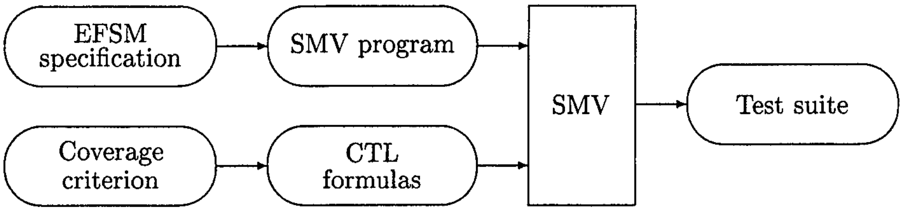


Figure 1: Overview of test generation

The remainder of the paper is organized as follows: After describing related work in Section 2, we give a brief introduction to CTL model checking in Section 3. We define the syntax and semantics of EFSMs in Section 4. We describe test coverage criteria and a test generation method for EFSMs in Sections 5 and 6, respectively. Finally, we conclude the paper with a description of future work in Section 7.

2 Related Work

Widely-used system models in the testing literature include finite state machines (FSMs) and labelled transition systems (LTSs), especially in hardware testing and protocol conformance testing. Testing methods based on such models primarily focus on the control-flow oriented test generation such as transition tour, unique-input-output sequence, distinguishing sequence, and characterizing sequence (for survey [3, 8, 17]). Although these methods are well-suited for hardware circuits and control portions of communication protocols, they are not powerful enough to test complex data-dependent behaviors.

EFSMs extend FSMs with variables to support the succinct specification of data-dependent behaviors. If the state space of an EFSM is finite, one can construct the equivalent FSM by unfolding the values of variables. Thus, EFSM-based testing with finite state space can be reduced in principle to ordinary FSM-based testing. Of course, this approach suffers from the well-known state explosion problem which makes test generation often impractical. Even when test generation is feasible, this approach is often impractical because of the test explosion problem, i.e., the number of generated tests might be too huge to be applied to implementations.

A promising alternative is to apply conventional software testing techniques to test generation from EFSMs [22]. In this approach, an EFSM is transformed into a flow graph that models the flow of both control and data in the EFSM and tests are generated from the graph by identifying the flow information. The approach abstracts the values of variables when constructing flow graphs and hence it can be applicable even if the state space is infinite. However, it requires posterior analysis such as symbolic execution or constraint solving to determine the executability of tests and for the selection of variable values which make tests executable.

The approach we advocate here is based on constructing Kripke structures from EFSMs, and thus, also suffers from the state explosion problem. However, the formulation of test generation as model checking in our approach enables the use of symbolic model checking [5] that has been shown to be effective for controlling the state explosion problem for certain problem domains. Second, our approach overcomes the test explosion problem by using control and data flow information of EFSMs like the flow-graph approach. Finally, our approach can be seen as complementary to the flow-graph approach. In particular, flow graphs can be constructed for systems whose state space is infinite, whereas our approach has the advantage that only executable tests are generated which obviates the need of posterior analysis. Ideally, one would eventually like to be able to combine these two approaches.

Recently connections between test generation and model checking has been considered in the literature. A tool that uses test generation methods inspired by model checking algorithms is described in [16]. Test generation using counterexamples constructed by a model checker has been applied in several contexts. Mutation analysis is used in the approach of [1]. In [6, 9], test generation is performed from user-specified temporal formulas. No consideration is given to coverage criteria. Several control flow coverage criteria are considered in [10]. We are not aware of any work that considers the model checking approach to data-flow oriented test generation.

3 CTL Model Checking

Symbolic model checking [5] is a commonly-used technique for the automatic verification of finite state systems. A widely-used temporal logic for symbolic model checking is CTL [7]. Let AP be the underlying set of atomic propositions. The syntax for CTL is defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi' \mid EX\phi \mid AX\phi \mid E[\phi U \phi'] \mid A[\phi U \phi']$$

where $p \in AP$ and ϕ, ϕ' range over CTL formulas. The remaining temporal operators are defined by the equivalence rules: $EF\phi \equiv E[trueU\phi]$; $AF\phi \equiv A[trueU\phi]$; $EG\phi \equiv \neg AF(\neg\phi)$; $AG\phi \equiv \neg EF(\neg\phi)$.

The semantics of CTL is defined with respect to a *Kripke structure* (S, S_0, L, R) where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; L: $S \rightarrow 2^{AP}$ is the state-labeling function;

and $R \subseteq S \times S$ is the set of transitions. A sequence s_0, s_1, s_2, ... of states is a *path* if $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. A path ρ is a s-path if $\rho(0) = s$. The satisfaction relation $\models$ is inductively defined as follows:

- $s \models p$ iff $p \in L(s)$; $s \models \neg\phi$ iff $\neg(s \models \phi)$; $s \models \phi \wedge \phi'$ iff $s \models \phi$ and $s \models \phi'$;
- $s \models EX\phi$ (resp. $AX\phi$) iff for some (resp. all) s-path ρ, $\rho(1) \models \phi$;
- $s \models E[\phi U \phi']$ (resp. $A[\phi U \phi']$) iff for some (resp. all) s-path ρ, there exists $i \geq 0$ such that $\rho(i) \models \phi'$ and $\rho(j) \models \phi$ for all $0 \leq j < i$.

An input to SMV consists of a symbolic representation of a Kripke structure as well as a list of CTL formulas to be checked. The symbolic representation consists of a set of variable declarations that determine the state space and a set of predicates over variables that describe the initial states and transition relation. Let V be a set of variables. We call v' as the primed version of a variable $v \in V$ and use V' to denote the set of primed versions of all variables in V. Let $\Sigma(V)$ be the set of all interpretations of V. We define a *SMV program* as a tuple (V, *Init*, *Trans*) where V is a finite set of variables; *Init* is a predicate on V; and *Trans* is a predicate on $V \cup V'$. A SMV program (V, *Init*, *Trans*) defines the Kripke structure (S, S_0, L, R) such that $S = \Sigma(V)$; $S_0 = \{\sigma \in \Sigma(V) \mid \sigma \models \mathit{Init}\}$; $L(\sigma) = \{v{=}\sigma(v) \mid v \in V\}$, for each $\sigma \in \Sigma(V)$; $(\sigma, \sigma') \in R$ if and only if $\langle \sigma, \sigma' \rangle \models \mathit{Trans}$, where $\langle \sigma, \sigma' \rangle$ is the interpretation that assigns $\sigma(v)$ to $v \in V$ and $\sigma'(v)$ to $v' \in V'$.

4 EFSMs

This section defines the syntax and semantics of EFSMs and introduces the notion of test sequences.

4.1 Syntax

An *extended finite state machine* (EFSM) is a tuple (Q, q_0, E, V, T) where Q is a finite set of *states*; q_0 is the *initial state*; E is a finite set of *events*, partitioned into E_I and E_O comprising *input* and *output* events, respectively. Each event is of the form $e(P)$ where e is an *event identifier* and P is a (possibly empty) set of *parameters*; V is a finite set of *variables*; T is a finite set of *transitions*, partitioned into T_I and T_O comprising *input* and *output* transitions, respectively. A transition is a tuple (q, $e(P)$, g, a, q') where $q \in Q$, $e(P) \in E$, and $q' \in Q$. If $e(P)$ is an input event, we require that g be a predicate on $V \cup P$ and a be a set of assignments to V. Otherwise, we require that g be a predicate on V and a is a set of assignments to $V \cup P$.

For a state q and event $e(P)$, define $G_{q,e(P)}$ as $\{g \mid (q, e(P), g, a, q') \in T\}$. An EFSM is *deterministic* if, for each state q and event $e(P)$, $g_i \wedge g_j = \mathit{false}$ for all $g_i, g_j \in G_{q,e(P)}$ such that $g_i \neq g_j$. In deterministic EFSMs, there is at most one possible transition to be taken at a given state and event. An EFSM is *completely specified* if, for each state q and event $e(P)$, $g_1 \vee ... \vee g_n = \mathit{true}$ where $G_{q,e(P)} = \{g_1, ..., g_n\}$. In completely specified EFSMs, there is at least one possible transition to be taken at a given state and event. For simplicity, this paper considers only deterministic and completely specified EFSMs.

We illustrate our approach using the bounded retransmission protocol (BRP), an extended version of alternating bit protocol that aborts transmission following a bounded number of retransmission attempts [14]. The protocol consists of a sender, a receiver, and two channels between them. This paper focuses on only the sender and adopts the EFSM specification for the sender by Rusu et al.[21] (see Figure 2).

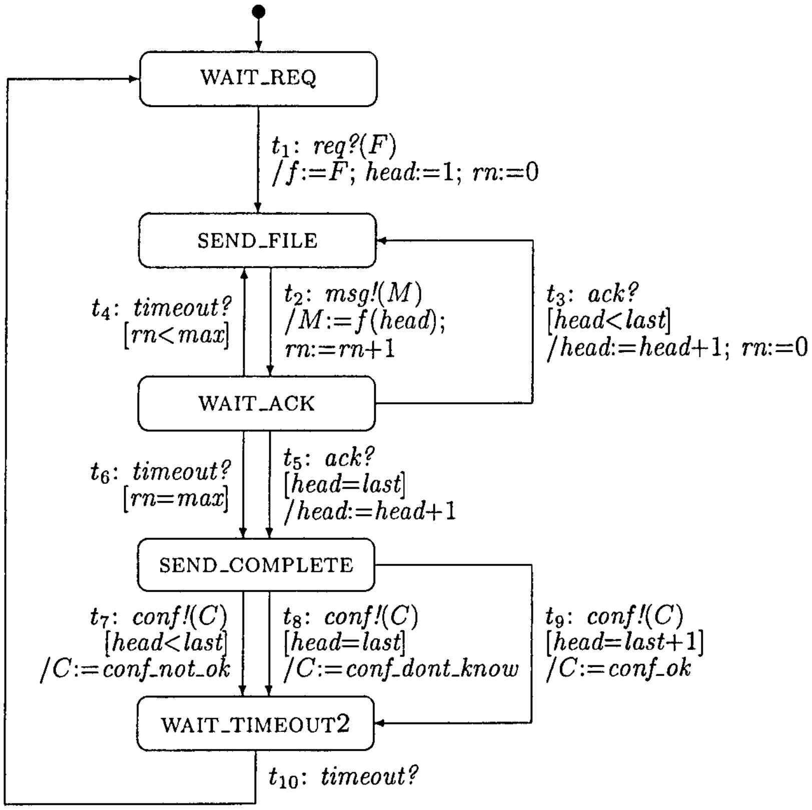


Figure 2: The sender of the BRP

The set of events in Figure 2 is partitioned into E_I = {*req?(F)*, *ack?*, *timeout?*, *timeout2?*} and E_O = {*msg!(M)*, *conf!(C)*}. The meaning of events is described by *req?(F)*: a list of messages F arrives, *ack?*: acknowledgment received, *msg!(M)*: a message M transmitted, *conf!(C)*: a confirmation C made. There are two timeout events denoted by *timeout?* and *timeout2?*. The set of variables is defined as {f, *last*, *head*, *max*, and *rn*}. The variable f is a list of messages, *last* is used to denote the length of f, and *head*, which is of integer subrange [1, *last*+1], indicates the message of f to be transmitted. *max* is the maximum number of retransmissions allowed for the sender, whereas *rn*, integer subrange [0, *max*], records the current number of retransmissions performed by the sender.

The EFSM's state space for the BRP sender is infinite due to (i) the type of messages, (ii) the length of message lists (*last*), (iii) the number of retransmissions (*max*). To make the state space finite, we scale down the EFSM by choosing the type of messages as boolean and by fixing *last* and *max* as 3 and 2, respectively. This enables the application of model checking to automatic test generation for the BRP sender. Of course this scaling-down is an ad-hoc abstraction and does not preserve all the possible behaviors of the original EFSM. We believe that abstractions techniques in the verification literature can be applied to constructing more general finite state abstractions from infinite EFSMs (see, for example, [12]).

4.2 Semantics

To generate tests from EFSMs using CTL model checking, we view EFSMs as Kripke structures. Because we use the symbolic model checker SMV, we do not enumerate Kripke structures explicitly but encode them symbolically using a set of variables and predicates. First we represent the state space of an EFSM (Q, q_0, E, V, T) in terms of (i) a variable π which ranges over Q; (ii) a boolean variable for each event identifier; (iii) a variable for each variable and parameter; (iv) a boolean variable for each transition.

The set of initial global states is described by the predicate *Init* as follows:

$$\mathit{Init} ::= (\pi{=}q_0) \wedge \bigwedge_{e\in E}(e{=}0) \wedge \bigwedge_{t\in T}(t{=}0)$$

which states that there is no event and transition taken prior to the initialization.

For an expression *exp*, we denote by $exp[V'/V]$ the result of replacing each occurrence of $v \in V$ in *exp* by its primed version v'. Finally we capture the transition relation of an EFSM as follows:

$\mathit{Trans} ::= \bigvee_{t\in T} \mathit{Trans}_t$

$\mathit{Trans}_t ::= \mathit{Enabled}_t \rightarrow \mathit{Taken}_t$

if t is an input transition and $a = \{v{:=}exp_v \mid v \in V\}$,

$\quad \mathit{Enabled}_t ::= (\pi{=}q) \wedge (g[P'/P]{=}1)$

$\quad \mathit{Taken}_t ::= (\pi'{=}q') \wedge (e'{=}1) \wedge \bigwedge_{e_1\in E, e_1\neq e}(e_1'{=}0) \wedge \bigwedge_{v\in V}(v'{=}exp_v[P'/P]) \wedge$
$\qquad (t'{=}1) \wedge \bigwedge_{t_1\in T, t_1\neq t}(t_1'{=}0)$

if t is an output transition and $a = \{v{:=}exp_v \mid v \in V\} \cup \{p{:=}exp_p \mid p \in P\}$

$\quad \mathit{Enabled}_t ::= (\pi{=}q) \wedge (g{=}1)$

$\quad \mathit{Taken}_t ::= (\pi'{=}q') \wedge (e'{=}1) \wedge \bigwedge_{e_1\in E, e_1\neq e}(e_1'{=}0) \wedge \bigwedge_{v\in V}(v'{=}exp_v) \wedge$
$\qquad \bigwedge_{p\in P}(p'{=}exp_p) \wedge (t'{=}1) \wedge \bigwedge_{t_1\in T, t_1\neq t}(t_1'{=}0)$

Note that there is no constraint on the next value of the parameters of input events because their changes are determined by the environment of the EFSM, whereas the parameters of output events are determined by the EFSM itself. Appendix A shows the SMV program corresponding to the EFSM in Figure 2.

4.3 Test Sequences

A *test sequence* of an EFSM is a finite path of its corresponding Kripke structure and a *test suite* is a set of test sequences. Figure 3 shows a test sequençe for the BRP sender which corresponds to the execution of the transition sequence t_1, t_2, t_3. For legibility, each global state shows only the events and transitions that are taken at that state.

We compare the nature of test sequences for reactive systems to those for transformational systems. Most of analysis and testing methods for transformational systems, e.g., flow graphs[13] and program dependency graphs[15], contain a distinguished node called exit node to model the terminating behaviors of such systems. Test sequences are naturally defined in terms of paths whose first node is the entry node and last node is the exit node of the graphs. On the other hand, there is no corresponding notion in reactive system models, because the behavior of reactive systems is characterized by their non-terminating executions.

A widely used approach to defining pseudo-exit nodes in EFSMs is to require that the execution of test sequences end at an initial state from which another test sequence can be applied. In

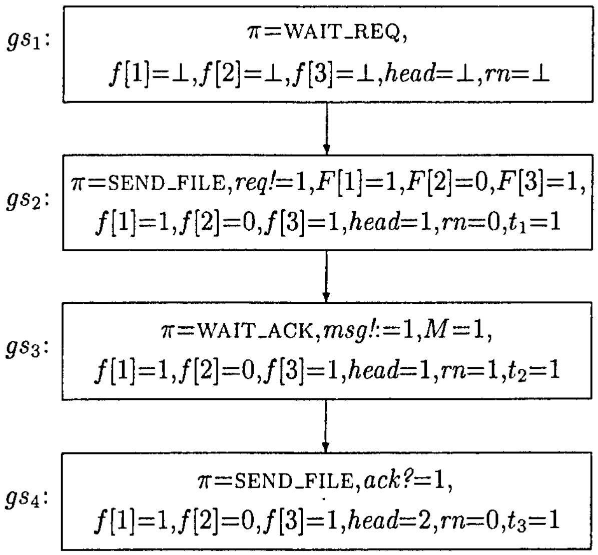

gs_1:
π=WAIT_REQ,
$f[1]=\perp, f[2]=\perp, f[3]=\perp, head=\perp, rn=\perp$
gs_2:
π=SEND_FILE, $req!=1, F[1]=1, F[2]=0, F[3]=1,$
$f[1]=1, f[2]=0, f[3]=1, head=1, rn=0, t_1=1$
gs_3:
π=WAIT_ACK, $msg!.=1, M=1,$
$f[1]=1, f[2]=0, f[3]=1, head=1, rn=1, t_2=1$
gs_4:
π=SEND_FILE, $ack?=1,$
$f[1]=1, f[2]=0, f[3]=1, head=2, rn=0, t_3=1$

Figure 3: Example of test sequences

general, testers may want to designate an arbitrary state as a pseudo-exit node. An interesting notion is marker state in the supervisory control theory by Ramadge and Wonham [19]. The theory distinguishes paths ending at a state designated as a marker from others and interprets such paths as completed tasks of the machine. For example, a tester may want to designate the state WAIT_REQ in Figure 2 as a marker and require that the execution of all test sequences end at the marker. In this case, the test sequence shown in Figure 3 can be extended by executing the transition sequence t_2, t_3, t_2, t_3, t_5, t_9, t_{10} in order to guarantee that the machine ends at WAIT_REQ.

5 Test Coverage Criteria for EFSMs

EFSMs specify the required behavior of implementations under test by describing the possible sequences of input and output events in terms of control and data dependencies between events. Specification-based testing with EFSMs aims at determining whether an implementation establishes the desired flow of both control and data expressed in its specification.

5.1 Control Flow Oriented Test Coverage Criteria

Obviously the strongest test coverage criteria for checking the conformance of an implementation to its EFSM specification is *path coverage* which requires that all paths of the EFSM's reachability graph be traversed. Because there is, in general, an infinite number of paths, it is impossible to achieve exhaustive testing and we need to have systematic coverage criteria that select a reasonable number of tests satisfying certain conditions. This paper investigates a family of test coverage criteria based on the flow information of both control and data expressed in EFSMs.

We say that a test sequence gs_0, gs_1, ..., gs_n *covers* a state q (resp. a transition t) if there exists i such that $gs_i(\pi) = q$ (resp. $gs_i(t) = 1$).

State coverage. A test suite P satisfies *state coverage* if every state in Q is covered by a test sequence in P.

Transition coverage. A test suite P satisfies *transition coverage* if every transition in T is covered by a test sequence in P.

5.2 Data Flow Oriented Test Coverage Criteria

We adopt the following convention which classifies each variable occurrence as being a definition or use. For a variable v and a transition $t = (q, e, g, a, q')$, v is *defined* at t if a contains an assignment that defines v and is *used* at t if a contains an assignment that references v or g references v. We use *def*(v) and *use*(v) to denote the sets of transitions that define and use v, respectively.

We discuss how structured variables such as arrays are handled, say f in Figure 2. In general it is not possible to determine statically the particular array element which is being defined or used. Therefore, an assignment to the variable $a[e]$ will consist of a definition of a and an use of each variable referenced at e. An use of $a[e]$ will consist of an use of a and an use of each variable referenced at e. Table 1 shows the information of definitions and uses for the BRP sender.

Table 1: The definitions and uses for the BRP sender

	f	*head*	*rn*
def	$\{t_1\}$	$\{t_1, t_3, t_5\}$	$\{t_1, t_2, t_3\}$
use	$\{t_2\}$	$\{t_2, t_3, t_5, t_7, t_8, t_9\}$	$\{t_2, t_4, t_6\}$

Let v be a variable and t, t' be transitions. We say that a path gs_0, gs_1, ..., gs_n is a *definition-clear path* with respect to v from t to t' if $gs_i(t)$=1, $gs_j(t')$=1, and there is no definition of v at t'' such that $gs_k(t'') = 1$, for each k in $i < k < j$. We say that a tuple (v, t, t') is a *def-use association* if $t \in def(v)$, $t' \in use(v)$, and there exists a definition-clear path with respect to v from t to t'.

Table 2 shows the def-use associations in Figure 2. For example, consider the tuple (*head*, t_1, t_3). It is a def-use association because the definition of *head* at t_1 can reach the use of *head* at t_3 through the definition-clear path shown in Figure 3. For another example, the tuple (*head*, t_1, t_5) is not a def-use association because there is no definition-clear path with respect to *head* from t_1 to t_5.

We say that a test sequence covers a def-use association (v, t, t') if the test sequence is a definition-clear path with respect to v from t to t'.

All-def coverage. A test suite P satisfies *all-def coverage* if for each variable v and each transition t such that $t \in def(v)$, some def-use association (v, t, t') is covered by a test sequence in P.

All-use coverage. A test suite P satisfies *all-use coverage* if for each variable v and each transition t such that $t \in def(v)$, every def-use association (v, t, t') is covered by a test sequence in P.

Table 2: The def-use associations for the BRP sender

variables	def-use associations
f	(f, t_1, t_2)
$head$	$(head, t_1, t_2)$, $(head, t_1, t_3)$, $(head, t_1, t_7)$ $(head, t_3, t_2)$, $(head, t_3, t_3)$, $(head, t_3, t_5)$, $(head, t_3, t_7)$, $(head, t_3, t_8)$, $(head, t_3, t_9)$, $(head, t_5, t_9)$
rn	(rn, t_1, t_2), (rn, t_2, t_2), (rn, t_2, t_4), (rn, t_2, t_6), (rn, t_3, t_2), (rn, t_3, t_4), (rn, t_3, t_6)

6 A Test Generation Method for EFSMs

This section shows that test generation from EFSMs can be automatically performed by using the ability of model checkers to construct counterexamples. Briefly the generation of a test suite from a given EFSM and test coverage criterion consists of the following steps.

- An SMV program for the given EFSM is constructed as described in Section 4.2.
- A set of CTL formula is constructed from the criterion.
- A test suite is constructed by model-checking each CTL formula against the SMV program.

6.1 Control Flow Oriented Test Generation

We represent each coverage criterion described in the previous section as a set of CTL templates. For a given EFSM, the set of templates is instantiated into a set of CTL formulas which captures exactly the coverage criterion for the EFSM.

We begin with the state coverage criteria which requires that for each state q, there exists at least one test sequence covering q. For a state q, we define the predicate $in(q)$ as $\pi{=}q$. Let $exit$ be a predicate defined as $in(marker)$ if there is a marker state $marker$ and $true$ otherwise. The Kripke structure for an EFSM has a path covering q if and only if (i) there exists a global state gs_i which is reachable from an initial global state gs_0 and at which $in(q)$ is satisfied; (ii) there exists a global state gs_j which is reachable from gs_i and at which $exit$ is satisfied (see Figure 4). We express this requirement using the CTL formula $EF(in(q) \wedge EF\,exit)$.

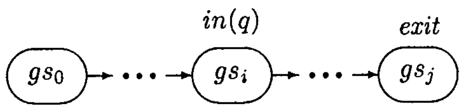


Figure 4: A test sequence covering state q

Now we take the negation of the above formula and run SMV against the negated formula $\neg EF(in(q) \wedge EF\,exit)$ because we are interested in generating test sequences covering q instead of checking the satisfiability of the original formula. If there exists a test sequence covering q, SMV generates a counterexample which corresponds to a test sequence covering q. Otherwise, SMV provides the result of *true* which implies that there is no test sequence covering q. There are two cases in which SMV provides *true* against the negated formula. First the global state gs_i is not reachable from any initial global state, i.e., $EFin(q)$ is not satisfied at any initial state. Second gs_j is not reachable from gs_i, i.e., $EF exit$ is not satisfied at gs_i.

To generate test sequences satisfying state and transition coverage from an EFSM $M = (Q, q_0, E, V, T)$, we use the following sets of CTL formulas.

State coverage. $\{\neg EF\,(in(q) \wedge EF\,exit) \mid q \in Q\}$

Transition coverage. $\{\neg EF\,(t \wedge EF\,exit) \mid t \in T\}$

6.2 Data Flow Oriented Test Generation

The following predicates are used to encode the information about definitions and uses of a variable v in SMV.

$$d(v) ::= \bigvee_{t \in def(v)} t \qquad\qquad u(v) ::= \bigvee_{t \in use(v)} t$$

For example, for the BRP sender, we have $d(f) ::= t_1$, $u(f) ::= t_2$, $d(head) ::= t_1 \vee t_3 \vee t_5$, $u(head) ::= t_2 \vee t_3 \vee t_5 \vee t_7 \vee t_8 \vee t_9$, $d(rn) ::= t_1 \vee t_2 \vee t_3$, and $u(rn) ::= t_2 \vee t_4 \vee t_6$.

The requirement for a def-use association (v, t, t') can be stated as follows: (i) there exists a global state gs_i which is reachable from an initial global state gs_0 and at which t is satisfied; (ii) there exists a path gs_{i+1} ... gs_{j-1} which starts from a successor of gs_i and contains no definition of v until gs_j at which t' is satisfied; (iii) there exists a global state gs_k which is reachable from the global state gs_j and at which *exit* is satisfied (see Figure 5). We express this requirement using $EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)])$.

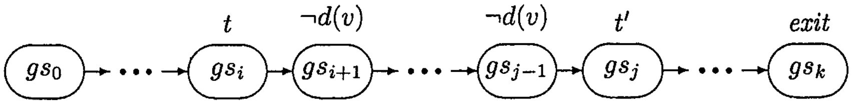


Figure 5: A test sequence covering def-use association (v, t, t')

For each tuple (v, t, t') such that $t \in def(v)$ and $t' \in use(v)$, we determine whether the tuple is a def-use association or not by associating the negation of the above formula $\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)])$ with the tuple. If SMV generates the result of *true* against the negated formula, the tuple is not a def-use association. Otherwise, the counterexample generated by SMV corresponds to a test sequence covering the def-use association (v, t, t').

All-def coverage.

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(u(v) \wedge EF\,exit)]) \mid v \in V, t \in def(v)\}$$

All-use coverage.

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)]) \mid v \in V, t \in def(v), t' \in use(v)\}$$

6.3 Example

From a given coverage criterion, we construct a set of CTL formulas and model check each formula in the set against the SMV program corresponding to an EFSM. If the formula is false, the model checker produces a counterexample which yields a test sequence to be included in a test suite. As an example, Appendix B shows the counterexample which is obtained by model checking $\neg EF(t_3 \wedge EFin(\textsc{wait\_req}))$, which is a formula from the transition coverage criterion formula set. This counterexample corresponds to the execution of event sequence *req?*(F[1]=0,F[2]=0,F[3]=0), *msg!*(M=0), *ack?*, *msg!*(M=0), *timeout?*, *msg!*(M=0), *timeout?*, *conf!*(*C*=*conf_not_ok*), *timeout2?*. Appendix C shows test suites for the BRP sender with respect to state, transition, all-def, and all-use coverage criteria, respectively.

7 Current and Future Work

We have presented a symbolic model checking approach to automatic test generation from EFSMs. Our approach considers a hierarchy of coverage criteria based on the information of both control flow and data flow in EFSMs and expresses each coverage criterion as a collection of CTL formulas. Tests are generated by finding counterexamples during model checking the formulas against EFSMs. The resulting test suite provides the capability of determining whether an implementation establishes the desired or required flow of control and data prescribed in its EFSM specification.

Other formalisms. Our representation of coverage criteria as collections of CTL formulas is language-independent and is applicable with minor modifications to any kind of specification languages based on EFSMs, e.g., SDL, Esetelle, and Statecharts. In fact, semantic differences in such languages affect only the translation method into input to SMV.

Other coverage criteria. A number of other coverage criteria based on control and data flow have been proposed in the software testing literature (see, for example, [20]). Some of these coverage criteria cannot be handled by SMV since it generates only one counterexample. For example, all-du-path coverage criterion requires that all definition-clear paths for each definition-use association be traversed. To generate tests for this criterion properly, we need all counterexamples to each CTL formula instead of only one. Such coverage criteria can be handled by extending SMV to construct multiple counterexamples for a CTL formula or by using a different model checker that has this capability.

Nondeterminism. In the case of non-deterministic EFSMs, there may be more than one possible output event sequence for a given input event sequence. In this situation, a single counterexample constructed by model checkers is not enough for the input event sequence, since it identifies only one output event sequence among all possible ones. One possible solution to this problem is to treat the counterexample as prescribing only the input event sequence. An extra step is then necessary to find all output event sequences corresponding to this input event sequence. If we have a model checker that produces multiple counterexamples to a formula, as discussed in the previous paragraph, we can express the input event sequence as a formula and give its negation to the model checker. The set of counterexamples constructed by the model checker will contain all possible output sequences.

Optimization. Often the test suite constructed by our approach will contain redundant tests. For example, the counterexample shown in Appendix B covers the transitions t_1, t_2, t_4, t_6, t_7, t_{10} in addition to t_3. It is, therefore, necessary to develop techniques to minimize the total number or the total length of generated tests without sacrificing the coverage they provide.

Test Execution. This paper is only the beginning and we are studying the above mentioned issues to make the approach practical. In addition, this paper focuses on only test generation and does not discuss the problem of test execution. As a first step, we are investigating how to apply generated tests to actual implementations under test. Our goal includes the development of automated toolsets which support both test generation and execution, and the evaluation of our approach on embedded systems.

References

[1] P. Ammann, P. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," in *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods*, pp. 46-54, 1998.

[2] F. Belina and D. Hogrefe, "The CCITT-Specification and Description Language SDL," *Computer Networks and ISDN Systems*, Vol. 16, pp. 311-341, 1989.

[3] G.v. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pp 109-124, 1994.

[4] S. Budkowski and P. Dembinski, "An Introduction to Estelle: a Specification Language for Distributed Systems," *Computer Networks and ISDM Systems*, Vol. 14, No. 1, pp. 3-24, 1991.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1990.

[6] J. Callahan, F. Schneider, and S. Easterbrook, "Specification-based Testing Using Model Checking," in *Proceedings of 1996 SPIN Workshop*, also Technical Report NASA-IVV-96-022, West Virginia Univeristy, 1996.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244-263, Apr. 1986.

[8] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test Development for Communication Protocols: towards Automation," *Computer Networks*, Vol. 31, pp. 1835-1872, 1999.

[9] A. Engels, L. Feijs, and S. Mauw, "Test Generation for Intelligent Networks Using Model Checking," in *Proceedings of TACAS '97*, Lecture Notes in Computer Science, Vol. 1217, pp. 384-398, Springer-Verlag, 1997.

[10] A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 6-10, 1999.

[11] D. Harel, "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[12] K. Havelund and N. Shankar, "Experiments in Theorem Proving and Model Checking for Protocol Verification," in *Formal Methods in Europe*, Lecture Notes in Computer Science, Vol. 1051, pp. 662-681, Springer-Verlag, 1996.

[13] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[14] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager, "Proof Checking a Data Link Protocol," in *Types for Proofs and Programs*, Lecture Notes in Computer Science, Vol. 806, pp. 127-165, Springer-Verlag, 1994.

[15] B. Korel, "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol. 24, pp. 103-108, Jan. 1987.

[16] T. Jeron and P. Morel, "Test Generation Derived From Model Checking," in *Computer Aided Verification '99*, Lecture Notes in Computer Science, Vol. 1633, pp. 108-121, Springer-Verlag, 1999.

[17] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, Vol. 84, No. 8, pp. 1090-1123, Aug. 1996.

[18] K.L. McMillan, *Symbolic Model Checking – an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[19] P.J. Ramadge and W.M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM Journal of Control and Optimization*, Vol. 25, No. 1, pp. 206-230, Jan. 1987.

[20] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367-375, Apr. 1985.

[21] V. Rusu, L. du Bousquet, and T. Jéron, "An Approach to Symbolic Test Generation," in *Proceedings of the International Conference on Integrating Formal Methods*, 2000.

[22] H. Ural, K. Saleh, and A. Williams, "Test Generation Based on Control and Data Dependencies within System Specifications in SDL," *Computer Communications*, Vol. 23, Issue 7, pp. 609-627, Mar. 2000.

A The SMV Program for the BRP Sender

```
MODULE main
VAR
-- states
pi: {WR, SF, WA, SC, WT};
-- events
req: boolean; ack: boolean; timeout: boolean; timeout2: boolean;
msg: boolean; conf: boolean;
-- parameters
F: array 1..3 of boolean; M: boolean; C: {conf-ok, conf-not-ok, conf-dont-know};
-- variables
f: array 1..3 of boolean; head: 1..4; rn: 0..2;
-- transitions
t1: boolean; t2: boolean; t3: boolean; t4: boolean; t5: boolean;
t6: boolean; t7: boolean; t8: boolean; t9: boolean; t10: boolean;
DEFINE
trans-t1 := enabled-t1 & taken-t1;
enabled-t1 := pi=WR;
taken-t1 := next(pi)=SF & next(req)=1 & next(ack)=0 & next(timeout)=0 & next(timeout2)=0 &
            next(msg)=0 & next(conf)=0 &
            next(f[1])=next(F[1]) & next(f[2])=next(F[2]) & next(f[3])=next(F[3]) &
            next(head)=1 & next(rn)=0 &
            next(t1)=1 & next(t2)=0 & next(t3)=0 & next(t4)=0 & next(t5)=0 &
            next(t6)=0 & next(t7)=0 & next(t8)=0 & next(t9)=0 & next(t10)=0;
trans-t2 := enabled-t2 & taken-t2;
enabled-t2 := pi=SF;
taken-t2 := next(pi)=WA & next(req)=0 & next(ack)=0 & next(timeout)=0 & next(timeout2)=0 &
            next(msg)=1 & next(conf)=0 &
            (head=1 -> next(M)=f[1]) & (head=2 -> next(M)=f[2]) & (head=3 -> next(M)=f[3]) &
            next(f[1])=f[1] & next(f[2])=f[2] & next(f[3])=f[3] &
            next(head)=head & next(rn)=rn+1 &
            next(t1)=0 & next(t2)=1 & next(t3)=0 & next(t4)=0 & next(t5)=0 &
            next(t6)=0 & next(t7)=0 & next(t8)=0 & next(t9)=0 & next(t10)=0;
trans-t3 := enabled-t3 & taken-t3;
enabled-t3 := pi=WA & head<3;
taken-t3 := next(pi)=SF & next(req)=0 & next(ack)=1 & next(timeout)=0 & next(timeout2)=0 &
            next(msg)=0 & next(conf)=0 &
            next(f[1])=f[1] & next(f[2])=f[2] & next(f[3])=f[3] &
            next(head)=head+1 & next(rn)=0 &
            next(t1)=0 & next(t2)=0 & next(t3)=1 & next(t4)=0 & next(t5)=0 &
            next(t6)=0 & next(t7)=0 & next(t8)=0 & next(t9)=0 & next(t10)=0;
.
.
.
INIT
pi=WR & req=0 & ack=0 & timeout=0 & timeout2=0 & msg=0 & conf=0 &
t1=0 & t2=0 & t3=0 & t4=0 & t5=0 & t6=0 & t7=0 & t8=0 & t9=0 & t10=0
TRANS
trans-t1 | trans-t2 | trans-t3 | trans-t4 | trans-t5 |
trans-t6 | trans-t7 | trans-t8 | trans-t9 | trans-t10
```

B The Counterexample for t_3

```
-- specification !EF (t3 = 1 & EF exit) is false
-- as demonstrated by the following execution sequence
state 1.1:
pi=WR req=0 F[1]=0 F[2]=0 F[3]=0 ack=0 timeout=0 timeout2=0 msg=0 M=0 conf=0 C=conf-dont-know
f[1]=1 f[2]=1 f[3]=1 head=4 rn=2
t1=0 t2=0 t3=0 t4=0 t5=0 t6=0 t7=0 t8=0 t9=0 t10=0

state 1.2:
pi=SF req=1 f[1]=0 f[2]=0 f[3]=0 head=1 rn=0 t1=1

state 1.3:
pi=WA msg=1 rn=1 t2=1

state 1.4:
pi=SF ack=1 head=2 rn=0 t3=1

state 1.5:
pi=WA msg=1 rn=1 t2=1

state 1.6:
pi=SF timeout=1 t4=1

state 1.7:
pi=WA msg=1 rn=2 t2=1

state 1.8:
pi=SC timeout=1 t6=1

state 1.9:
pi=WT conf=1 C=conf-not-ok t7=1

state 1.10:
pi=WR timeout2=1 t10=1

resources used:
user time: 0.26 s, system time: 0.03 s
BDD nodes allocated: 10365
Bytes allocated: 1376256
BDD nodes representing transition relation: 566 + 9
```

C Test Suites for the BRP Sender

State coverage

state	test sequence
WAIT_REQ	ϵ
SEND_FILE	*req?*(0,0,0)
WAIT_ACK	*req?*(0,0,0), *msg!*(0)
SEND_COMPLETE	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*
WAIT_TIMEOUT2	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*not_ok*)

Transition coverage

transition	test sequence
t_1	*req?*(0,0,0)
t_2	*req?*(0,0,0), *msg!*(0)
t_3	*req?*(0,0,0), *msg!*(0), *ack?*
t_4	*req?*(0,0,0), *msg!*(0), *timeout?*
t_5	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *ack?*
t_6	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*
t_7	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*not_ok*)
t_8	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*dont_know*)
t_9	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *ack?*, *conf!*(*ok*)
t_{10}	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*ok*), *timeout2?*

All-def coverage

definition	test sequence
(f, t_1)	*req?*(0,0,0), *msg!*(0)
$(head, t_1)$	*req?*(0,0,0), *msg!*(0)
$(head, t_3)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0)
$(head, t_5)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *ack?*, *conf!*(*ok*)
(rn, t_1)	*req?*(0,0,0), *msg!*(0)
(rn, t_2)	*req?*(0,0,0), *msg!*(0), *timeout?*
(rn, t_3, t_2)	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0)

All-use coverage

tuple	test sequence
(f, t_1, t_2)	*req?*(0,0,0), *msg!*(0)
$(head, t_1, t_2)$	*req?*(0,0,0), *msg!*(0)
$(head, t_1, t_3)$	*req?*(0,0,0), *msg!*(0), *ack?*
$(head, t_1, t_5)$	*infeasible*
$(head, t_1, t_7)$	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*not_ok*)
$(head, t_1, t_8)$	*infeasible*
$(head, t_1, t_9)$	*infeasible*
$(head, t_3, t_2)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0)
$(head, t_3, t_3)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*
$(head, t_3, t_5)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *ack?*
$(head, t_3, t_7)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*not_ok*)
$(head, t_3, t_8)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *msg!*(0), *timeout?*, *conf!*(*dont_know*)
$(head, t_3, t_9)$	*infeasible*
$(head, t_5, t_2)$	*infeasible*
$(head, t_5, t_3)$	*infeasible*
$(head, t_5, t_5)$	*infeasible*
$(head, t_5, t_7)$	*infeasible*
$(head, t_5, t_8)$	*infeasible*
$(head, t_5, t_9)$	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0), *ack?*, *msg!*(0), *ack?*, *conf!*(*ok*)
(rn, t_1, t_2)	*req?*(0,0,0), *msg!*(0)
(rn, t_1, t_4)	*infeasible*
(rn, t_1, t_6)	*infeasible*
(rn, t_2, t_2)	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0)
(rn, t_2, t_4)	*req?*(0,0,0), *msg!*(0), *timeout?*
(rn, t_2, t_6)	*req?*(0,0,0), *msg!*(0), *timeout?*, *msg!*(0), *timeout?*
(rn, t_3, t_2)	*req?*(0,0,0), *msg!*(0), *ack?*, *msg!*(0)
(rn, t_3, t_4)	*infeasible*
(rn, t_3, t_6)	*infeasible*

A C interface to the Concurrency Workbench

Daniel C. DuVarney[1] [2]
W. Rance Cleaveland[3][4]
S. Purushothaman Iyer[1][2]

Abstract

Software model-checking aims to apply model-checking techniques, which have been found effective in reasoing about finite state designs, to programs. An integral part of this process is the generation of a finite state model from programs. In this paper we report on how C-programs can be abstracted and reasoned about in the Concurrency Workbench in a seemless manner, thus yielding a C-interface to the workbench. A further advantage of our approach is that designs that are checked for correctness in the Workbench can now be related to programs that implement those designs.

1 Introduction

The past twenty years have seen great strides in the use of model-checking to validate finite state designs of programs and circuits. Given the impressive results that have been achieved [5] in these limited applications there have been efforts to apply model-checking to software systems [6, 11] and to infinite state designs [2, 4]. The former style of work is based on abstracting programs to arrive at finite state designs, which can then be model-checked. The latter, however, depends upon considering decidable properties of certain subclasses of infinite state systems. In this paper we address the approach of abstraction.

Apart from model-checking software we are also motivated by another problem: that of relating designs and implementations. In traditional software life cycles a (finite-state) design may be constructed and validated with respect to requirements stated in a formal, logic-based language. However, any implementation of the design is not checked against the validated design. Our thesis is that abstract interpretation can play a huge role in bridging the gap between designs and implementations. More importantly, by constructing a (finite-state) abstraction of an implementation we can (a) model-check software and (b) relate designs and implementations using notions of refinement and simulation [12]. In the rest of the paper we will discuss our efforts in building the tool C2CCS that allows an user to abstract C-programs, and reason about the abstraction (and, in turn, the original program) using the Concurrency Workbench of the New Century (CWB-NC) [15]. In particular, C2CCS strives to make the abstraction process and the reasoning process seamless so that, in effect, an user deals with a C-interface to the CWB-NC.

An user, typically, provides to the tool C2CCS a C-program with additional guidance information, such as events to observe, variables to abstract, etc. The tool C2CCS produces a finite state abstraction of the input program as a Labeled Transition System – the main internal data-structure for CWB-NC. Furthermore, an additional tool makes it possible to view the LTS by translating it to the format of daVinci [10], a system for viewing graphs. The ability to visualize an LTS provides valuable feedback to an user who can fine-tune his/her abstraction. The tool makes it unnecessary to manually construct a design of a system (as, for instance, in [9]), but instead derive it from the source code.

The main contributions of our work are the following

1. An user-driven abstraction process where the user dictates what events are relevant, how variables have to be abstracted, how inter-procedural analysis should be carried out, and how function pointers should be interpreted.

2. Given that we are interested in generating a finite state design from an input program we restrict our attention to tail-recursive programs. The model-building, i.e., generation of an LTS, is carried out so that the structure of the original program is preserved in the abstraction. More importantly, the control structure of the original program is maintained in the translation thus making it easy for the user to correlate the graphical view of the abstraction against the original program.

[1] Dept of Computer Science, North Carolina State University, Raleigh, NC 27695-7534.

[2] Supported in part by ARO grant P-38682-MA.

[3] Dept of Computer Science, State Univeristy of New York at Stony Brook, Stony Brook, NY 11794-4400

[4] Supported by AFOSR grant F49620-95-1-0508, ARO grant P-38682-MA and NSF grants CCR-9505562, CCR-9996086 and INT-9996095.

```
main () {
 ...

 P();
 ...
}
```

```
int P() {
  ..

  if (p) {
     ...
      Q();
    }
  else {
     ...
   }
 return ;
```

```
int Q() {
  ..
  if (r) {
    ...
     P();
   }
   else {
      ...
   }
  return
```

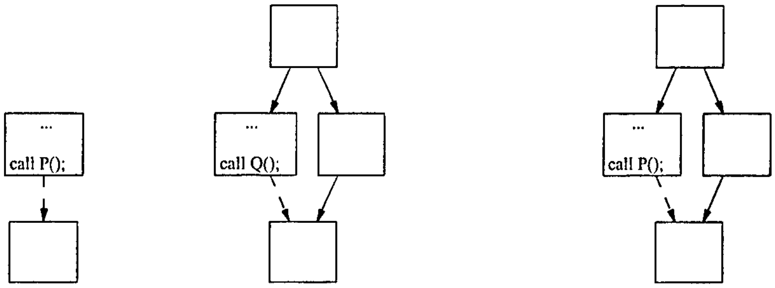


Figure 1: An example program and its conrol-flow graph

3. We generate an LTS directly from a program which is then passed on to the reasoning algorithms of the Concurrency workbench. This allows us to seemlessly integrate abstraction and reasoning. To achieve this effect we make use of CWB-NC's ability to deal with any state space representation as long as it comes equipped with certain operations.

4. A graphical representation of the design that is generated from the program.

Related work There are several projects that have recently considered using abstraction to generate models from programs [7, 11, 3]. In the project Bandera [6], Corbett *et al* use *slicing* to generate a finite-state abstraction of the original program – where the slicing (i.e. what variables to ignore) is dictated by the property that is to be established. Holzmann and Smith [11] allow an user to provide patterns for transforming C-code to an abstracted version in Promela. Finally, Brylow, Damhaard and Palsberg [3] abstract assembly code to control flow graphs with procedure calls embedded in them. Furthermore, they use recent work on model-checking pushdown systems and its connection to flow analysis [4, 14].

2 Early experiments: The dataless case

Given a C-program it can be encoded in the Calculus for Communicating Systems [13] as CCS is Turing-powerful. However, we would not be able to reason about the translated program. Clearly, the translation/abstraction should be in a decidable subclass such that it captures the behavior of the original program and some of its relevant properties too. As a first cut we will discuss what it means to ignore all the data values, and abstract the control structure. The sequence of procedure calls that are made will be observable in the abstraction and will be a superset of actual sequences of the program. Such an abstraction will preserve universally path quantified temporal formulae, but is seldom useful in practise. We will, nevertheless, discuss the salient points of this abstraction as a stepping stone for discussing our current implementation.

Imagine translating a C-program to intermediate code, as is typically done in compilers. Such a translation would break long expression calculations to a sequence of single-operation calculations, and will also translate control structures such as `while` and `for` statements into tests and `goto`s. From the resulting translation

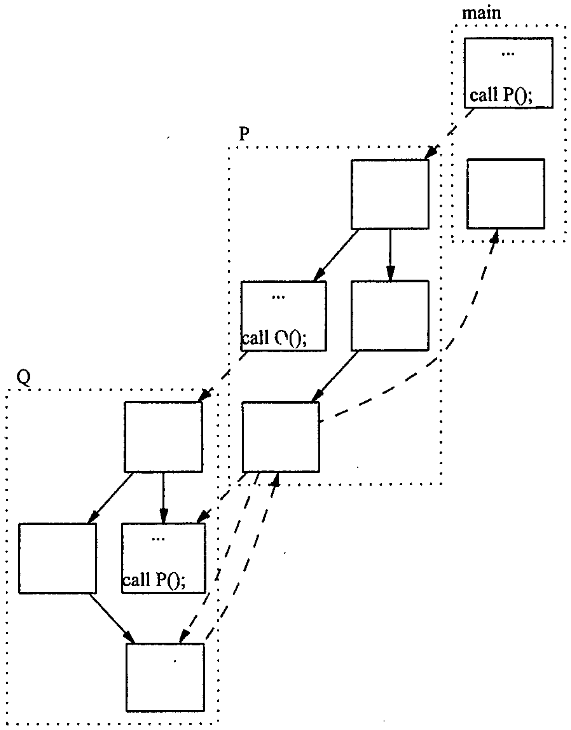


Figure 2: A transition system for the program of figure 1

one can extract a *control-flow* graph [1] where each node is a piece of straight-line code terminated by either a goto or a procedure call. We construct such a graph for each of the procedures in the input program. Given that we wish to translate our C-programs to a finite state Labeled Transition System we will have to restrict ourselves to tail-recursive C-programs.

To determine whether a given C-program is tail-recursive or not we consider the *call graph* of the input program, where nodes are procedures and edges denote procedure calls (with source being the caller and target, of the edge, being callee). Clearly, a procedure that appears in a strongly connected component of such a graph indicates that it could call itself recursively, though perhaps in an indirect fashion.

A particular call to procedure Q in procedure P is said to be a *tail* call provided P returns immediately after control returns from procedure Q, i.e., there are no other calculations in procedure P once Q returns. A strongly connected component is said to be *tail-recursive* provided each call in the procedures of the strongly connected component are tail calls. A program is said to be tail-recursive provided all of the strongly connected components are tail-recursive. Note that this is a syntactic property, not a semantic one, and can be easily checked.

Note that procedure call and return are indicated textually and using dashed-arrows, respectively.

Assume we are given a C-program that is tail recursive. To build an LTS for it we start with control-flow graphs for each of the procedures, such as in Figure 1. Furthermore, notice that the call-graph for this program has procedures P and Q in the only (non-trivial) strongly connected component of the graph. Choosing one of this SCC, say P, as a representative and expanding (inlining) calls to other procedures the strongly connected component can be converted into a single function. In this case it involves substituting a copy of Q's control-flow graph in graph for P. With repeated substitutions, if necessary, we get a single control-flow graph of a function that can be converted to iteration (because of its tail-recursiveness). We present the results of these substitutions for our example in Figure 2.

Domain	Representation	Description
Unity	None	All values abstracted to $\top$.
Mod k	Bit-vector ($2 \cdot k + 1$ bits)	Values are abstracted to the set of possible remainders when divided by k.
Interval $x_1 \ldots x_n$	Bit-vector ($2 + n$ bits)	Given a set of break-points $x_1 \ldots x_n$ ($x_1 < x_2 < \ldots < x_n$), each value is abstracted to a subset of the set of intervals $(\{[\infty, x_1), [x_i, x_{i+1})](1 \leq i < n), [x_n, +\infty]\})$ to which it might belong.

Table 1: Abstract domains.

```
type A_index = part(0, 1, 2, 98, 99, 198);

file "example.c"
{
  fun binsearch(var l : A_index; var h : A_index; var x : top) : top
  {
    var m : A_index;
  }

  fun find() : top
  {}
}
```

Figure 3: An abstraction mapping file for the source file of Figure 4

3 Incorporating Data and Abstractions

In addition to constructing the control flow graph our tool C2CCS also allows an user to incorporate abstractions of the input program's data in the finite state design. The tool is not fully automatic but provides an user-driven abstraction process where the user specifies how to abstract the program data.

The abstraction mechanism currently supports only integer values. Table 1 lists the abstract domains. There are three basic kinds of abstraction. The first is the unity abstraction, which is the coarsest possible abstraction wherein all concrete values are mapped to the same abstract value. The second is the modulo k abstraction, in which a set of remainders is maintained relative to a user-specified parameter k. The third is the interval abstraction, in which integer values are partitioned into contiguous subranges by the user, and the abstract value is a bit-vector indicating which subrange(s) the concrete value might be in. The motivation for the "modulo" abstraction comes from the common use of finite counters in protocols, such as the sliding window protocol.

The user controls how abstractions are applied. The initial input is an abstraction mapping file, which gives an abstraction for each variable and each function in the program from which a model is to be extracted. An example mapping file appears in Figure 3. If a variable or function is omitted from the mapping file, the default abstraction is the unity abstraction.

For each abstract domain the system has an abstraction function α which maps from concrete to abstract values, a (unimplemented) concretization function γ which maps from abstract to concrete values, a least upper bound operator $\sqcap$, and abstract implementations of the C operators.

All our domains are classic abstract interpretations as defined by [8]. The values in each domain form a lattice, with $\top$ representing all possible concrete values and $\bot$ representing non-terminating computations (note: $\bot$ is currently unused).

Within the lattice, the $\sqcup$ operator merges two abstract values. $x \sqcup y$ returns a value z such that $\gamma(x) \subseteq \gamma(z)$ and $\gamma(y) \subseteq \gamma(z)$). $\sqcup$ is used to collect a single abstract value from multiple execution paths in a manner similar to traditional data-flow analysis.

The abstract operators are all provably sound. Specifically, $\gamma(\alpha(x) \circ \alpha(y)) \subseteq x \circ y$ has been proved for

```
int A[100];

int binsearch(int l, int h, int x)
{
  int m = (l + h) / 2;
  if (A[m] == x) return m;
  if (l >= h) return -1;
  if (A[m] > x) return binsearch(l, m-1, x);
  return binsearch(m+1, h, x);
}

int find(int x) {
  return binsearch(0, 99, x);
}
```

Figure 4: C program to be abstracted.

	Unity	Mod$_{k_1}$	Interval$_{x_1,x_2,\ldots x_n}$	Precise
Unity	Unity	Unity	Unity	Unity
Mod$_{k_2}$	Unity	Mod$_{g}cd(k_1, k_2)$	Mod$_{k_2}$	Mod$_{k_2}$
Interval$_{y_1,y_2,\ldots y_n}$	Unity	Mod$_{k_1}$	Interval$_{\{x_1,\ldots x_n\}\cup\{y_1\ldots y_n\}}$	Interval$_{y_1,y_2,\ldots y_n}$
Precise	Unity	Mod$_{k_1}$	Interval$_{x_1,x_2,\ldots x_n}$	Precise

Table 2: Rules for combining abstractions.

each operator $\circ$. Furthermore, definitions of the operators have been chosen which generally lose as little information as possible.

When values from different abstract domains are combined in an operation, there is an implicit coercion to the more abstract domain. The hierarchy of domains is: Interval $\sqsubset$ Mod $\sqsubset$ Unity. Table 2 gives the full rules for combining mixed abstractions. In general, mixing abstractions is not very useful, as it often results in a complete loss of information.

With the definition of abstract operators in place, the effect of a *basic block* can be determined. Basic blocks correspond to nodes in the control-flow graph and consist of a sequence of non-branching statements followed by at most one branching statement (conditional branch, unconditional branch, or case statement). A basic block is evaluated in the context of an *environment* in which all variables have been bound to an abstract value. Given an environment e, the result of executing a basic block under e is a set of environment and block pairs. This set is computed by evaluating all the non-branching statements to yield an environment e', and then applying any constraints that can be inferred from the branch taken to get multiple environment and block pairs. For example, consider the block

```
x = x - 1;
y = y - x;
if x > 0 then goto block2 else goto block3;
```

Further, assume that x and y are abstracted to the domain $2^{\{neg,zero,pos\}}$ and that the initial environment is:

$$[x \mapsto \{pos\}, y \mapsto \{neg\}]$$

Now evaluation of the non-branching statements `x = x - 1` and `y = y - x` results in the environment:

$$[x \mapsto \{zero, pos\}, y \mapsto \{neg\}]$$

The block / environment pairs resulting from evaluation of the branch on `x > 0` in this environment are:

$$\{\langle block2, [x \mapsto \{pos\}, y \mapsto \{neg\}]\rangle, \langle block2, [x \mapsto \{zero\}, y \mapsto \{neg\}]\rangle\}$$

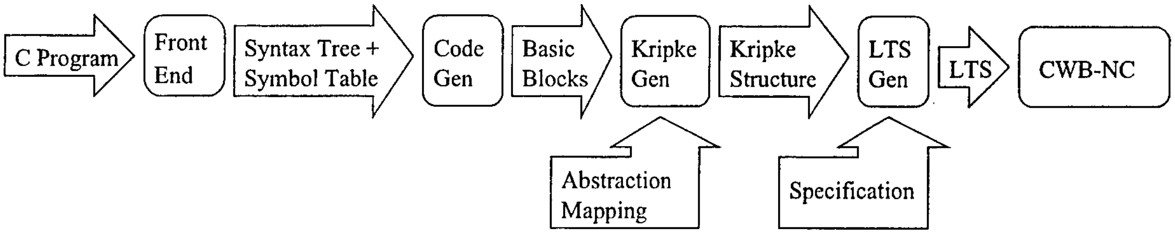


Figure 5: Data flow within the system architecture.

```
let
    $B$ = the set of basic blocks
    $E : B \rightarrow Env$ map from basic blocks to environments.
    $b_0$ = first block of main
    $e_0$ = initial environment ($e(v) = \alpha$(inital value of $v$) for all $v$)
    $e_\epsilon$ = the empty environment
in
    $E(b) = \begin{cases} e_0 & \text{if } b = b_0 \\ e_\epsilon & \text{otherwise} \end{cases}$
    $P := \{b_0\}$
    while ($P \neq \emptyset$)
        pick any $b \in P$
        $P := P - \{b\}$
        let $e = E(b)$
        let $succ(b, e)$ be the successors of $\langle b, e \rangle$
        for each $\langle b_i, e_i \rangle \in succ(b, e)$ do
            let $e'_i = E(b_i) \sqcup e_i$
            if $e'_i \neq E(b_i)$ then
                $P := P \cup \{b_i\}$
                $E := (E - \{\langle()b_i, e_i)\}) \cup \{\langle b_i, e'_i \rangle\}$
```

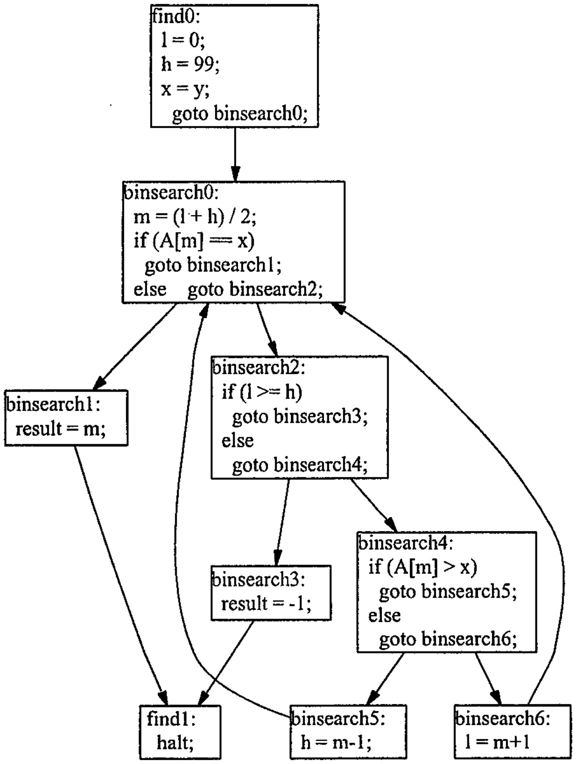
Figure 6: Monovariant Kripke structure generation.

4 Generation of the State Space

The state space generation for a system is essentially a four step process, as illustrated by Figure 5. First, the input files are parsed and compiled into control-flow graphs. In this process the abstraction to be applied to each function and variable is also recorded in the symbol table. Second, all function calls are unrolled, eliminating the control stack. Third, a Kripke structure is built by executing the unrolled code in the abstract domain(s) specified by the user. Finally, the Kripke structure is converted to an LTS.

The first step taken by the system is to parse the abstraction mapping file. This file gives the pathnames of all the C source files in the program. A high-level intermediate code is generated, organized into basic blocks.

In the second step, function calls are unrolled, as discussed in Section 2. The intermediate code allows for basic blocks to be *aliased*, wherein the alias block contains a pointer to the original block plus a new branching instruction which supercedes the last instruction in the original block. Although function-call unrolling may cause a large increase in the size of the intermediate code, we believe aliasing will reduce memory usage to the point of practicality for many programs.



```
find0:
 l = 0;
 h = 99;
 x = y;
  goto binsearch0;

binsearch0:
 m = (l + h) / 2;
 if (A[m] == x)
   goto binsearch1;
 else   goto binsearch2;

binsearch1:
 result = m;

binsearch2:
 if (l >= h)
   goto binsearch3;
 else
   goto binsearch4;

binsearch3:
 result = -1;

binsearch4:
 if (A[m] > x)
   goto binsearch5;
 else
   goto binsearch6;

find1:
 halt;

binsearch5:
 h = m-1;

binsearch6:
 l = m+1
```

Figure 7: Program of Figure 4 after basic block generation and function call unrolling.

4.1 Kripke Structure Generation

The third step is the generation of a Kripke structure. The Kripke structure states will be basic blocks and environment pairs. The environment will also serve as the label of the state.

We use two algorithms for generation of the Kripke structure. The first algorithm, shown in Figure 6, is monovariant over the values of variables, with every block in the unrolling corresponding to one state in the Kripke structure. It generates a single environment for each basic block, resulting in a smaller model of limited usefulness. Given a basic block b, an environment is collected over all execution paths which reach b. Abstract interpretation is then used to determine the environment(s) which are propagated to the successor blocks of b. This process is repeated until a fixpoint is reached for the structure (i.e., until the input environment for each block stabilizes).

Consider the program of Figure 4. One property we would like to check is whether or not the array accesses done by `binsearch` are guaranteed to be within the bounds of the array `A`. This can be done by generating a monovariant Kripke structure under the interval abstraction:

$$[-\infty \ldots -1], [0], [1], [2], [3 \ldots 98], [99], [100 \ldots 198], [199 \ldots +\infty]$$

After basic blocks have been generated and function calls unrolled, the result is the program of Figure 7. Applying the monovariant algorithm results in the kripke structure of Figure 8. From the kripke structure, we can infer that the value of `m` is within the bounds of the array `A`.

Polyvariant kripke structure generation results in a larger but more powerful model. The polyvariant algorithm, shown in Figure 9, generates a seperate state for each combination of basic block and environment. The polyvariant algorithm also uses abstract interpretation, but when multiple execution paths cause different environments to be input to a basic block, the environments are not combined. Instead, a separate states is generated for each unique block and environment pair. This generates many more states but also enables many more properties to be verified.

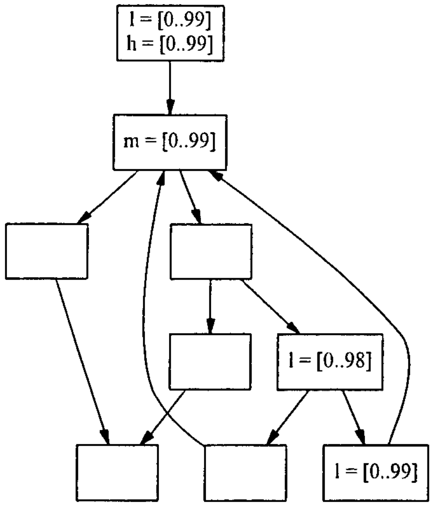


Figure 8: Monovariant Kripke structure generated from program of figure 7.

```
let
    $B$ = the set of basic blocks
    $b_0$ = first block of main
    $e_0$ = initial environment ($e(v) = \alpha$(initial value of $v$) for all $v$)
    $Q = Q' = \{\langle b_0, e_0 \rangle\}$
in
    while ($Q' \neq \emptyset$)
        pick any $\langle b, e \rangle \in Q'$
        $Q' := Q' - \{\langle b, e \rangle\}$
        Let $succ(b, e)$ be the successors of $\langle b, e \rangle$
        $New = succ(b, e) - Q$
        $Q := Q \cup New$
        $Q' := Q' \cup New$
```

Figure 9: Polyvariant Kripke structure generation.

The program fragment of figure 10 is an example of when polyvariance is required to verify certain properties. Assume we want to know that the calls to P and V are strictly interleaved — that is, after calling P, V must be called before P is called again, and vice-versa. While both algorithms generate sound models, only the polyvariant model can be used to prove that calls to P and V are indeed interleaved. The second half of figure 10 shows the polyvariant LTS.

Memory usage is a critical issue, particularly under the polyvariant model. To conserve memory, all the environments are combined into a trie. Each node of the trie represents a particular variable. Each outgoing edge represents a binding of the variable to a particular value. The bindings encountered along any path from the root of the trie to a leaf define an environment. We hope this will conserve a large amount of memory, and plan to evaluate the utility of this storage technique in the near future.

4.2 Labeled Transition System Generation

The generation of a labeled transition system (henceforth called LTS) from a Kripke structure is done by discarding the labels on the states and attaching action labels to the transitions. The default label is the

```
int flag = 0;

while(!eof()) {
  int cmd = get_command();

  if (cmd == 1) {
    if (!flag) {
      P();
      flag = 1;
    }
  } else {
    if (flag) {
      V();
      flag = 0;
    }
  }
}
```

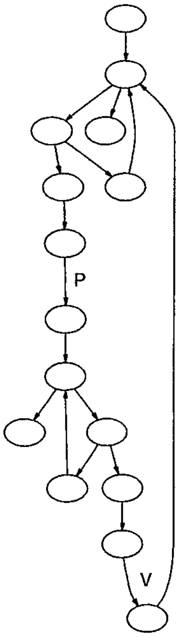


Figure 10: C program fragment and the resulting polyvariant LTS

invisible action τ. Visible actions are generated in two cases. The first is the case when an "interesting" function or line of code is executed. The second is the case when a particular condition is true, such as `0 <= m && m <= 99`. The user specifies which labels to associate with each event.

5 Conclusion and current state of the tool

Our tool is still a work in progress. Currently, the tool generates Kripke structures and displays them using the graph visualization tool /refDaVinci. Furthermore, we only use polyvariant model generation as it is most likely to yield usable abstractions of programs.

We have presented a tool that can be used to slice and dissect a C-program, and to reason about it. This is in sharp contrast to testing, which takes a black-box view of a system. Furthermore, by combining a model-generation tool and a reasoning tool, we have created a easily-usable front-end to the Concurrency Workbench. Finally, we have made it possible to compare designs against implementations.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Bouajjani, A., Echahed, R., and Habermehl, P. Verifying infinite state processes with sequential and parallel composition. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)* (San Francisco, California, Jan. 22–25, 1995), ACM Press, pp. 95–106.

[3] Brylow, D., Damgaard, N., and Palsberg, J. Static checking of interrupt-driven software. In *International Conference on Software Engineering* (2001).

[4] Burkart, O., and Steffen, B. Model-checking the full-modal mu-calculus for infinite sequential processes. In *Automata, Languages and Programming (ICALP '97)* (Bologna, Italy, July 1997),

P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds., vol. 1256 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 419–429. Full version to appear in *Theoretical Computer Science.*

[5] Clarke, E. M., and Wing, J. M. Formal methods : state of the art and future directio ns. *ACM Computing Surveys 28*, 4 (Dec. 1996), 626–643.

[6] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering* (Limerick, Ireland, June 2000), IEEE Computer Society, pp. 439–448.

[7] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering* (Limerick, Ireland, June 2000), IEEE Computer Society, pp. 439–448.

[8] Cousot, P. Abstract interpretation. *ACM Computing Surveys 28*, 2 (June 1996), 324–328.

[9] Dong, Y., Du, X., Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Sokolsky, O., Stark, E. W., and Warren, D. S. Fighting livelock in the i-protocol: A comparative study of verification tools. In *TACAS 99* (1999), pp. 74–88.

[10] Fröhlich, M., and Werner, M. Demonstration of the interactive graph-visualization system davinci. In *Graph Drawing* (Oct. 1994), R. Tamassia and I. G. Tollis, Eds., vol. 894 of *Lecture Notes in Computer Science*, DIMACS, Springer-Verlag, pp. 266–269. ISBN 3-540-58950-3.

[11] Holzmann, G. J., and Smith, M. H. Software model checking - extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems* (Kluwer Academic Publ., Oct. 1999), pp. 481–497.

[12] Larsen, K. G., and Xinxin, L. Equation solving using modal transition systems. In *Fifth Annual Symposium on Logic in Computer Science (LICS '90)* (Philadelphia, June 1990), IEEE Computer Society Press, pp. 108–117.

[13] Milner, R. *Communication and Concurrency.* PHI Series in Computer Science. Prentice Hall, 1989.

[14] Podelski, A. Model checking as constraint solving. In *SAS'00* (2000), no. 1824 in LNCS, pp. 22–37.

[15] R. Cleaveland, and S. Sims. The NCSU concurrency workbench. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV* (New Brunswick, NJ, USA, July/Aug. 1996), Rajeev Alur and Thomas A. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 394–397.

Specification of a Parallelizing SequenceL Compiler [1]

by
Daniel E. Cooke and Per Andersen
Computer Science Department
Texas Tech University
Lubbock, TX 79409

ABSTRACT
SequenceL is a language that provides declarative constructs for nonscalar processing. Rather than specifying program control structures that, in turn, imply a data product, the problem solver specifies a data product and the control structures to produce or process the data product are implied. Although *SequenceL* has been previously introduced in two papers [2, 3], recent improvements to the language have indicated that parallel control structures are also implied by the *SequenceL* problem solutions. **Keywords:** High Level Language, Automatic Parallelisms, Executable Specifications

1.0 A brief review of SequenceL.

One of the first major advances in computer language design, often called the von Neumann architecture, was the elimination of the distinction between program and memory. This advance eventually led to the paradigms where data is stored in named locations and algorithms can define the locations through the use of assignment and input-output statements. Advances also led to the development of the control flow constructs that are used to define and process data structures. These constructs are the **sequence**, **selection**, **iteration**, and **parallel** constructs.

The *SequenceL* paradigm [2, 3] provides no distinction between data and functions/operators. Furthermore, all data is considered to be nonscalar, i.e., sequences. Recently, an underlying control flow used in the evaluation of *SequenceL* operators has been introduced [4]. In the underlying **Consume-Simplify-Produce**-cycle (i.e., **CSP**-cycle), operators enabled for execution are **Consumed** (with their arguments) from the global memory, **Simplified** according to declarative constructs that operate on data, and the resulting simplified terms are then **Produced** in the global memory. This **CSP**-cycle, together with the declarative *SeqenceL* constructs (applied during the simplification step) imply control flow structures that the programmer does not have to design or specify.

The elimination of the distinction between data and operators is achieved through the use of a global memory, called a tableau *T*, where *SequenceL* terms are placed:

$$[f^{a}1_{1}, f^{a}2_{2}, \ldots, f^{a}n_{n}] \text{ where } n \geq 0$$

when $a_i = 0$ f_i is a sequence and when $a_i > 0$ f_i is a function or an operator. The arity a_i of a function or operator f_i indicates the number of arguments (i.e., sequences) f_i requires for execution. There is no notion of assignment in *SequenceL*. Furthermore, there is no notion of input distinguished from the application of an operator to its operands. In this context, the word **operator** refers both to built-in and user-defined operators.

The underlying **CSP** control flow in *SequenceL* eliminates the need for the programmer to specify parallel and most of the specification of iterative and sequence-based operations. The control flow approach identifies operators that are enabled for execution. The enabling of operators is based upon the arity of the operator (which is derived from the operator's signature) matching the number of sequences following the operator. For example, suppose the global memory *T* contains the following:

$$T = [f^{2}_{1}, f^{3}_{1}, f^{0}_{1}, f^{0}_{2}, f^{0}_{3}, f^{2}_{2}, f^{0}_{4}, f^{0}_{5}]$$

Where, $[f^{3}_{1}, x, y, z] = x$, $[f^{2}_{2}, x, y] = y$, and $[f^{2}_{1}, x, y] = xy$. The control flow is with reference to *T* and involves a repeated cycle of consuming, simplifying, and producing. The state of the machine is reflected – not with program counters, register and memory states, etc. – but by the global memory alone. As long as there exist **enabled** operators in the memory, execution proceeds. In *T*, above, there are two enabled operators, namely f^{3}_{1} and f^{2}_{2},. These operators and their arguments

[1] Research sponsored, in part, by NASA NAG 5-9505.

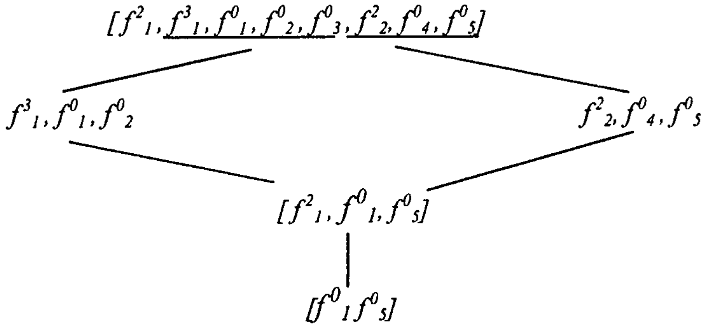
are **consumed** from the tableau, and they execute in parallel. Placeholders mark in *T* where respective results will be placed. Execution **simplifies** the consumed terms. Suppose the simplified terms result in a final state for the function f^3_1 namely f^0_1 and a final state for f^2_2,, namely f^0_5. The resulting tableau contains the simplifications **produced** as a result of the execution of the operators f^3_1 and f^2_2:

$$T' = [f^2_1, f^0_1, f^0_5]$$

Now f^2_1 executes, and simplification leads to a sequence to concatenated terms:

$$T'' = [[f^0_1 f^0_5]$$

The final Tableau (seen above) can be viewed as the "output" of the program of operations. The total computation is reflected in the following graph, where the vertical dimension indicates a sequence of simplification steps (i.e., the sequence control flow construct) and the horizontal dimension indicates actions that can take place in parallel (i.e., the parallel construct). Thus, sequential and parallel structures are derived through the application of the built-in **CSP**-cycle to *T:*



However, CSP alone will not achieve the desired effect. In order to derive finer-grained parallelisms and many iterative processes in an automated way, CSP must interact with advanced constructs for processing data. The desired interaction occurs in the **Simplification** step of the **CSP**-cycle. These advanced *SequenceL* constructs are the **regular**, **generative**, and **irregular** constructs.

The regular construct applies an operator to corresponding elements of the normalized operand sequences. For example, *+([4,4])* distributes the plus sign among the corresponding elements of the two singleton sequences *[4]* and *[4]*, resulting in *[8]*. Now consider the more complicated application:

$$T = [*([20,30,40],[2])]$$

In this case normalization results in the elements of the second operand sequence being repeated in the order they occur until it is the same length (and/or dimension in terms of nesting) as the larger:

$$T = [*([20,30,40],[2,2,2])]$$

Now the multiplication operator can be distributed among the corresponding elements of the operand sequences. The normalization/distribution is a single **simplification** step:

$$T' = [*([20,2]),\ *([30,2]),\ *([40,2])]$$

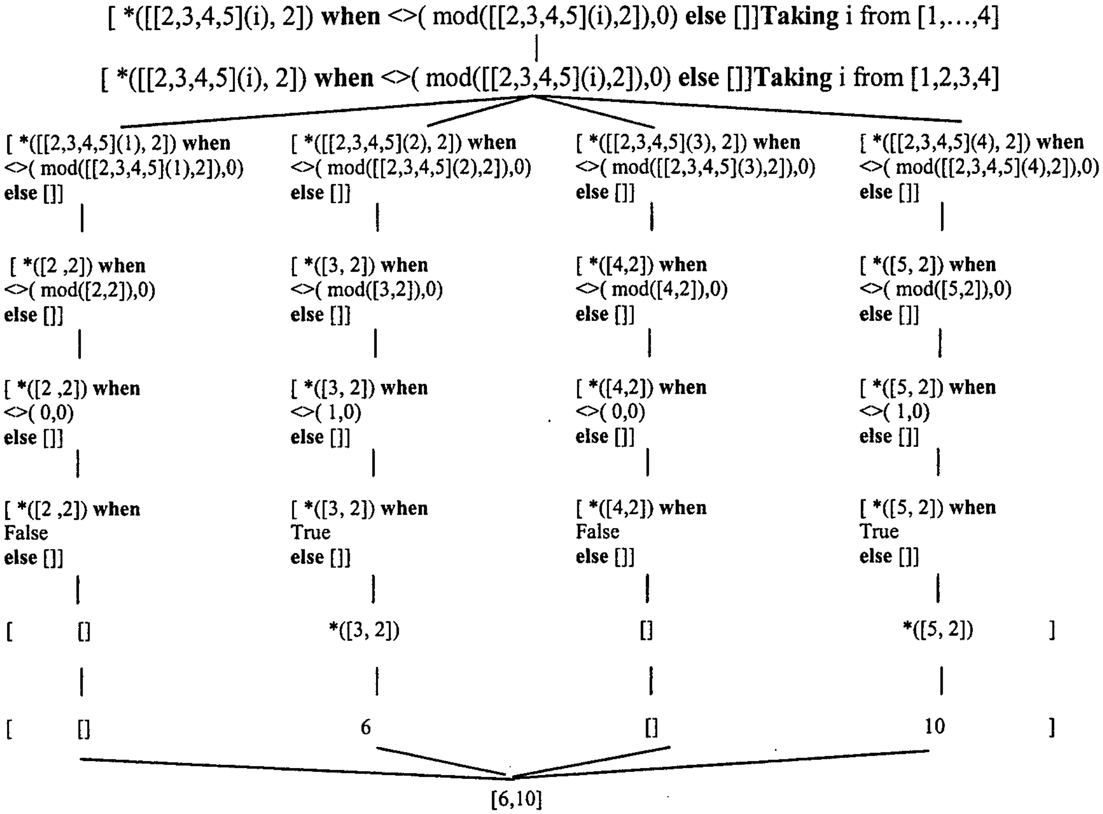
As a result, in the next **CSP** cycle there are three parallel operations that can take place. Thus, the interaction of the CSP execution cycle and the regular construct results in the identification of microparallelisms. The final result is:

$$T'' = [40,\ 60,\ 80]$$

The generative construct allows for the generation of sequences, e.g., [1,...,5] = [1,2,3,4,5]. The irregular construct allows for conditional execution:

T=[Double-odds(**Consume**(s_1(n)), **Produce**(next)) where next =
[*([s_1(i) 2]) **when** <>(mod([s_1(i),2]),0) **else** []] **Taking** i from [1,...,n],[2,3,4,5]]

Here is the trace of the function:



[*([[2,3,4,5](i), 2]) **when** <>(mod([[2,3,4,5](i),2]),0) **else** []]**Taking** i from [1,...,4]

[*([[2,3,4,5](i), 2]) **when** <>(mod([[2,3,4,5](i),2]),0) **else** []]**Taking** i from [1,2,3,4]

[*([[2,3,4,5](1), 2]) **when** <>(mod([[2,3,4,5](1),2]),0) **else** []]

[*([[2,3,4,5](2), 2]) **when** <>(mod([[2,3,4,5](2),2]),0) **else** []]

[*([[2,3,4,5](3), 2]) **when** <>(mod([[2,3,4,5](3),2]),0) **else** []]

[*([[2,3,4,5](4), 2]) **when** <>(mod([[2,3,4,5](4),2]),0) **else** []]

[*([2 ,2]) **when** <>(mod([2,2]),0) **else** []]

[*([3, 2]) **when** <>(mod([3,2]),0) **else** []]

[*([4,2]) **when** <>(mod([4,2]),0) **else** []]

[*([5, 2]) **when** <>(mod([5,2]),0) **else** []]

[*([2 ,2]) **when** <>(0,0) **else** []]

[*([3, 2]) **when** <>(1,0) **else** []]

[*([4,2]) **when** <>(0,0) **else** []]

[*([5, 2]) **when** <>(1,0) **else** []]

[*([2 ,2]) **when** False **else** []]

[*([3, 2]) **when** True **else** []]

[*([4,2]) **when** False **else** []]

[*([5, 2]) **when** True **else** []]

[[] *([3, 2]) [] *([5, 2])]

[[] 6 [] 10]

[6,10]

With the irregular construct one can, when necessary, produce recursive applications of a function:

T=[list(**Consume**(s_1), **Produce**(next)) where next =
[[list, -([s_1, 1])] **when** >(s_1,0) **else** [s_1]] [4]

Recursion involves a function placing itself back into T. Here is the trace of the function:

[list(**Consume**(s_1), **Produce**(next)) where next =
[[list, -([s_1, 1])] **when** >(s_1,0) **else** [s_1]] [2]

[[list, -([2, 1])] **when** >(2,0) **else** [2]]

[list(**Consume**(s_1), **Produce**(next)) where next =
[[list, -([s_1, 1])] **when** >(s_1,0) **else** [s_1]] [1]

```
                              |
   [ [list, -([1, 1])]  when >(1,0) else [1]]
                              |
[list(Consume(s_1), Produce(next)) where next =
     [ [list, -([s_1, 1])]  when >(s_1,0) else [s_1]] [0]

                              |
   [ [list, -([0, 1])]  when >(0,0) else [0]]
                              |
                             [0]
```

SequenceL, represents an attempt to develop a language abstraction in which control structures – sequential, iterative, and parallel – are implied.

In summary, the *SequenceL* abstraction presents.a paradigm with the following aspects:

- a Global Memory that does not distinguish between operators and data, and whose state fully reflects the state of computation;
- All operands are sequences where atoms are represented by singleton sequences;
- Underlying **Consume-Simplify-Produce** execution strategy; and
- High level constructs to process nonscalars, applied in the simplification step.

For more detail about *SequenceL* please see [2, 3]

2.0 SequenceL Interpreter

A *SequenceL* intepreter was constructed in 1999, based upon the **Consume-Simplify-Produce** control flow. A number of experiments were conducted to determine adept *SequenceL* is in finding inherent parallelisms in a problem definition. Included in these experiments was the Matrix Multiply[2]. Consider the matrix multiply function as written in *SequenceL*:

```
Function matmul(Consume(s_1(n,*),s_2(*,m),Produce(next)) where next = {compose([+([*(succ_1(i,*),succ_2(*,j))])])}
        Taking [i,j]From cartesian_product([gen([1,...,n]),gen([1,...,m])])
([[2,4,6],[3,5,7],[1,1,1]], [[2,4,6],[3,5,7],[1,1,1]])
```

The term, *[[2,4,6],[3,5,7],[1,1,1]]* is the sequence representation of a matrix. We will now trace the steps the *SequenceL* interpreter takes in order to evaluate the function above. Bear in mind the only information provided by the user is the function above, together with its arguments. The user does not specify any control structures – not even the parallel paths that can be followed in solving the problem. These parallel paths are implied in the solution and derived by the *SequenceL* semantic. The semantic is an interactions between the CSP-cycle and the *SequenceL* constructs applied in the **Simplify** step.

2.1 The Interpreter's Trace of the Evaluation of the SequenceL Matrix Multiply.

For simplicity, let *T* contain only the matrix multiply function and its input sequences as seen above. The first step is to instantiate the variables *s_1* and *s_2.* At the same time, variables *n* and *m* obtain the cardinal values in the designated sequence dimensions:

```
{compose([+([*([[2,4,6],[3,5,7],[1,1,1]](i,*), [[2,4,6],[3,5,7],[1,1,1]](*,j))])])}
        Taking [i,j]From cartesian_product([gen([1,...,3]),gen([1,...,3])])
```

Now *SequenceL's* **generative construct** produces the values needed in the **Taking** clause. The simple form of the generative command is seen in this example. It simply fills in the integer values between the upper and lower bounds, i.e., from *1* to *3:*

[2] For the classes of problems included in the full range of experiments please see [4].

```
{compose([+([*([[2,4,6],[3,5,7],[1,1,1]](i,*), [[2,4,6],[3,5,7],[1,1,1]](*,j))])])}
        Taking [i,j]From cartesian_product([[1,2,3],[1,2,3]])
```

Next the *SequenceL* Cartesian Product generates the values for subscripts *i* and *j*.

```
{compose([+([*([[2,4,6],[3,5,7],[1,1,1]](i,*), [[2,4,6],[3,5,7],[1,1,1]](*,j))])])}
        Taking [i,j]From [ [[1], [1]], [[1], [2]], [[1], [3]],
                           [[2], [1]], [[2], [2]], [[2], [3]],
                           [[3], [1]], [[3], [2]], [[3], [3]]          ]
```

Now that the simplification of the function is complete, the function's result is produced in *T:*

```
T = [
      [     +([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]     ||
      [     +([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]     ||
      [     +([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])       ||
            +([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]
    ]
```

The next *Consume-Simplify-Produce* (CSP) step replaces the tableau above with the one below. This simplification step results in the parallel selection of the vectors to be multiplied. Concurrent evaluation is denoted by the || symbol.

```
T =     [
              [     +([*([[2,4,6],[2,3,1]])])               ||
                    +([*([[2,4,6],[4,5,1]])])               ||
                    +([*([[2,4,6],[6,7,1]])])     ]         ||
              [     +([*([[3,5,7],[2,3,1]])])               ||
                    +([*([[3,5,7],[4,5,1]])])               ||
                    +([*([[3,5,7],[6,7,1]])])     ]         ||
              [     +([*([[1,1,1],[2,3,1]])])               ||
                    +([*([[1,1,1],[4,5,1]])])               ||
                    +([*([[1,1,1],[6,7,1]])])     ]
        ]
```

In the next CSP step, the products are formed concurrently using *SequenceL's* **regular construct**. The regular process distributes a built-in operator, e.g., the * operation, among corresponding elements of the operands, resulting in *27* parallel multiplies

```
T =     [
              [     +([*([[2], [2]]) || *([[4], [3]]) || *([[6], [1]])])         ||
                    +([*([[2], [4]]) || *([[4], [5]]) || *([[6], [1]])])         ||
                    +([*([[2], [6]]) || *([[4], [7]]) || *([[6], [1]])])   ]     ||
              [     +([*([[3], [2]]) || *([[5], [3]]) || *([[7], [1]])])         ||
                    +([*([[3], [4]]) || *([[5], [5]]) || *([[7], [1]])])         ||
                    +([*([[3], [6]]) || *([[5], [7]]) || *([[7], [1]])])   ]     ||
              [     +([*([[1], [2]]) || *([[1], [3]]) || *([[1], [1]])])         ||
                    +([*([[1], [4]]) || *([[1], [5]]) || *([[1], [1]])])         ||
                    +([*([[1], [6]]) || *([[1], [7]]) || *([[1], [1]])])   ]
        ]
```

Comparing the tableau above and the resulting tableau below, one can see that *SequenceL* handles nested parallelisms automatically. The final step of simplification involves the application of the regular construct to form the sums of the products.

```
T =     [      [      +([4,12,6])      //
                      +([8,20,6])      //
                      +([12,28,6]) ]   //
               [      +([6,15,7])      //
                      +([12,25,7])     //
                      +([18,35,7]) ]   //
               [      +([2,3,1])       //
                      +([4,5,1])       //
                      +([6,7,1]) ]             ]
```

This regular process adds together corresponding elements of the operand sequences. E.G., *+([4,12,6]) = [4+12+6] = [22]*. The final result is:

[[22, 34, 46], [28, 44, 60], [6, 10, 14]]

Since no further simplification is possible, evaluation ends. There are three *SequenceL* constructs: the regular, irregular, and generative. The matrix multiply employed the regular and the generative. For more detailed explanation of these constructs see [2, 3]

One important aspect of this language is that through the introduction of new language constructs, one can imply most control structures – even concurrent or parallel structures. Therefore, rather than producing an algorithm that implies a data product, one can come closer to specifying a data product that implies the algorithm that produces or processes it. The difficult part of traditional forms of programming seems to be centered around the fact that programmers have to somehow envision the elusive data product implied by their programs.

3.0 SequenceL Compiler.

The trace of the interpreter's execution of the Matrix Multiply is summarized graphically in figure 1. Much of the breadth and depth of any lattice structure outlined in the execution of a *SequenceL* function is very dependent upon the data presented to the functions and operations. The data for a program is not typically known prior to execution of the program. To contend with the unknown data at compile time, languages require programmers specify the type, size, and dimensions of data in their programs.

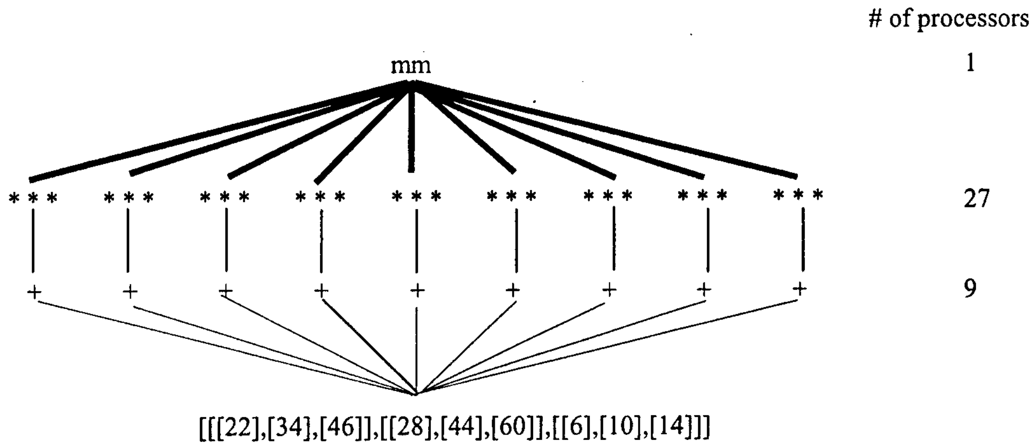


Figure 1. Trace of Matrix Multiply.

We have made a design decision that we would like to maintain the current language, which does not require this information. Of course, this is not an issue in building an interpreter, but does require extra thought in building a compiler that achieves our goal of generating parallelized multithreaded C code to run in a shared memory multiprocessing environment. Below is the full *SequenceL* grammar for which we will construct the compiler:

A → integer | real | string
L → A,L | T0,L | A | T0
E → [ ] | [L] | s(integer)
V → id
O → + | - | / | * | abs | sqrt | cos | sin | tan | log | mod | reverse | transpose | rotateright | rotateleft | cartesianproduct
M → *,M | T0,M | * | T0
T → E | V | O(T0) | gen([T0,...,T0]) | gen([T0,...,T0] by T0 | $
T0 → T | T(M) | T(map(M))
R0 → < | > | = | >= | <= | <> | integer | real | var | operator
R → R0(T0) | and(R^+) | or(R^+) | not(R^+)
B → $T0^+$ | $T0^+$ when R else B
C → [ ] | taking [V^+] from T0
F → V(V^+) where next = {B} C
U → V ,U | E, U | V | E
P → {{F^+ } U }

To contend with the unknowns involving the data to which a function will be applied, we are separating the compiler and scheduler issues in terms of the construction of an execution graph. In what follows we will focus on the evaluation of the multiplication and addition operators, i.e., the evaluations that take place subsequent to the instantiation of variables and the evaluation of the *taking* clause. Therefore, the graph in figure 1 is our focal point.

Notice that the normalization and distribution of operators is completely dependent upon the input variables. We can, therefore imagine the basic compiled outline of the matrix multiply to be:

... → ***Normalization* → *Distribution 1* → *Distribution 2* → *Distribution 3*** →
Multiplication → *Addition*

The outline could indeed be produced by a compiler – based upon the following distribution/evaluation rule:

Def. 1. $\Sigma(\Theta^{math} ([t_1, t_2,..., t_n])) =$

if $\mu' ([t_1 , t_2 ,... t_n]) = 1$ *then*

$[(t_1\ \Theta^{rmath} (t_2\ \Theta^{rmath} \ldots \Theta^{rmath} (t_{n-1}\ \Theta^{rmath}\ t_n)\ldots))]$

else

if $\mu' ([t_1 , t_2 ,... t_n]) = 2$ *then*

$[\Theta(t_1), \Theta(t_2), ..., \Theta(t_m)]$

where $[t_1, t_2,..., t_m] = transpose(v([t_1, t_2,..., t_n]))$

else

$[\Sigma(\Theta(t_1)), \Sigma(\Theta(t_2)), ..., \Sigma(\Theta(t_m))]$

where $[t_1, t_2,..., t_m] = v([t_1, t_2,..., t_m])$ *and* $\Theta^{math} \in \{ \wedge, +, -, *, /, \textbf{div}, \textbf{mod} \}$ *and v is the normalization operation and* Θ^{rmath} *is the real or actual mathematics operation corresponding to the* Θ^{math} *operator and* μ' *gives the level of nesting of the operand*

Consider the following example in which there are *3* levels of nested sequences. This example, where $\mu'([t_1, t_2, \ldots t_n]) = 3$, exercises all three of the **distributive** operations:

+([[[1,2,3],[4]], [[5,6,7],[5]], [[8,9,],[6]]]) =(3)
normalization and distribution

[+([[1,2,3],[4,4,4]]), +([[5,6,7],[5,5,5]]), +([[8,9,8],[6,6,6]])] =(2)
normalization (which has no effect), transposition, and distribution

[[+([1,4]), +([2,4]), +([3,4])], [+([5,5]), +([6,5]), +([7,5])], [+([8,6]), +([9,6]), +([8,6])]] =(1)
final distribution

[[[1+4], [2+4], [3+4]], [[5+5], [6+5], [7+5]], [[8+6], [9+6], [8+6]]]

In reviewing the tableau trace in section 2.1 and the lattice in figure 1, parallel threads of execution are apparent (e.g., in each descending branch of the lattice). The current plan is to compile computational components (e.g., the multiplication and addition computations) and have the scheduler perform the related normalization and distribution actions to result in the appropriate sequencing and parallelizing of computations. Notice that there is an interaction between the result of the inner products and the summation components of the matrix multiply. A stepwise scheduler that performs the normalization, distribution, and load balancing appears to be the part of the compiler that will require the mostattention and effort.

3.1 Technical Issues Related to Compilation and Scheduling.

The compiler for SequenceL will attempt to take advantage of the tableau structure which to date has provided many of the insights into SequenceL, specifically the implied parallelisms. Early on in the evaluation of the language and possible compiler design scheduling was recognized as playing a central role in the success of the compiler for parallel or multi-processor architecture. The scheduling of a task on a parallel computer will consider a number factors such as task granularity, task allocation and task synchronization. Granularity is a defined as the ratio between computation and communication [8] or how much computation should or will take place before there is communication between tasks. A fine-grained parallelism has very few instructions between communication cycles while a coarse grained parallelism has many instructions between communication cycles. Therefore the scheduler must be able to manage granularity. For example given a fine grained parallelism involving 1000 calculations the scheduler might create only 10 threads with 100 computations in each in order to keep inter-processor communications overhead at a minimum, for a coarse grained parallelism of 1000 calculations the scheduler might create 1000 threads of execution. Task allocation is closely tied to the computer's architecture, the scheduler might have to consider the differences between a non-uniform memory access (NUMA) machine and a uniform memory access machine when scheduling tasks.

Many of the more recent successful parallel architectures have been either shared memory architectures like that found in the Silicon Graphics Inc's Origin2000 which is a NUMA cache coherent system [6], or distributed memory systems such as IBM's SP and Beowulf clusters. Ideally any new parallel languages and compilers developed for those languages would be implemented for both of these architectures. For this first prototype parallel SequenceL compiler, only one of these architectures will have be chosen for the initial development. Typically, distributed memory architectures are more difficult to program since they are message passing architectures also they have fewer sophisticated development tools such as good parallel debuggers. Shared memory architecture, like the Origin2000, are shipped with vendor supplied integrated development tools and compilers that can take advantage of all processors using standard sequential programming techniques. In addition systems like the Origin2000 can also be programmed using message passing if necessary.

An early decision was made to implement the compiler using a subset of C as an intermediate programming language. Therefore, with this in mind and given the decision to use a shared memory architecture the choices for the underlying C based parallel development tool was narrowed down to three choices.

1) Message Passing Interface (MPI)
2) OpenMP
3) Multi-threading (Pthreads)

When it came to choosing one of these three parallel programming models the following criteria was used, control over task scheduling, ease in data sharing and portability.

MPI is a great parallel programming tool for the experienced parallel programmer. The availability of low-level message passing calls, provides the developer with a tool that gives them complete control over their parallel application. Scheduling tasks or threads of computation within a parallel application is not a problem since MPI gives the developer complete control over all of the tasks at a process level of execution. It is also the low-level nature of the MPI calls that also makes it a difficult development tool to use [1] in addition the process model of execution makes it a poor choice for fine grained parallelisms. Another factor against MPI was the fact that experience has shown that MPI is not available on many symmetrical multi-processor systems (SMP), it is typically found only on parallel computers as a vendor supplied package or as an add-on for cluster computers [5].

OpenMP is describes as a shared memory or distributed shared memory parallel programming tool [1]. Its implementation is at a higher level of abstraction than MPI. On the Origin2000 parallel program developers can utilize OpenMP in one of two ways. Sequential programs can be submitted to an OpenMP tool, which automatically adds OpenMP compiler directives to the code or a developer can manually place the directives in the code themselves. OpenMP directives are designed to take advantage of parallelisms in loops, it is not designed for functional parallelisms. MPI can take advantage of both functional and loop parallelisms. Scheduling with OpenMP is possible at a computational level with respect to loop iterations, but scheduling of specific processors is not available using a scheduling protocol.

The final parallel programming model evaluated was Pthreads. Pthreads is similar to MPI in that much of the coding of the parallel application is the responsibility of the developer. But it does have several important advantages over MPI, first it uses a thread model while MPI uses a process model, therefore task overhead could be less. Second it can take advantage of shared memory accesses, different threads can access the same memory locations, data sharing with MPI requires message passing. This could also be a disadvantage since it can lead to memory bottlenecks [7]. Although OpenMP is also based on a thread model and has the same shared memory access capability it does not provide the same level of control over thread scheduling that Pthreads does. Typical shared memory programming models follow a coarse grained thread model, Pthreads provide a better range of control over computational threads since they can be fine grained [9]. Finally unlike MPI and OpenMP a Pthreads based program will run without change on a single processor system as well as on parallel system like the Origin2000. The benefit therefore of implementing the compiler with Pthreads is that in developing a parallel compiler for SequenceL a single processor compiler will also result.

Given a SequenceL source code program and a final Pthreads based C program what are the required intermediate stages to get from one to the other? An early prototype utilized syntax changes, although the results were successful changes were required in the language in order to assist in the semantic analysis using this technique [10]. It was concluded that a better job of reduction based on the SequenceL simplification rules might have resulted in avoiding the language changes. The next prototype will use a parser/reducer that will simplify the source into a reduced SequenceL based on language simplification rules. From the reduced source code a translator will convert the source into a graph(s) structure. The justification for choosing a graph structure is the SequenceL tableau. It is not clear yet where this will lead, but the tableau data structure has proven to be a remarkably good tool in describing the parallelisms, which fall out of SequenceL. For example a simple 3x3 dense matrix multiplication has the following tableau structure.

```
T = [
            [     Σ+([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](1,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]
            [     Σ+([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](2,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]
            [     Σ+([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,1))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,2))])
                  Σ+([*([[2,4,6],[3,5,7],[1,1,1]](3,*),[[2,4,6],[3,5,7],[1,1,1]](*,3))]) ]
      ]
```

From this structure it is very easy to see that each line represents a parallel thread of execution. Within each thread is the computational task consisting of a multiply/add operation. The simplified graph representation of the above tableau could consist of computations going in the depth direction and scheduling issues going in the breath direction. The key therefore in going from SequenceL to C code will be the implementation of the data-flow and control flow in some form of graph representation that will provide the same level of description as the above tableau. Therefore the prototype parallel SequenceL compiler will have two intermediate representations of the SequenceL source code, the graph(s) representation and the C code.

Several key elements of a compiler have not been mentioned yet. One key stage in any compiler is optimization, for the first prototype parallel compiler for SequenceL optimization will be left to the C compiler. The only optimization that will occur in the prototype compiler will be related to thread scheduling.

4.0 Concluding Remarks.

SequenceL is a Turing complete language. An interpreter exists that finds the parallel structures inherent in *SequenceL* problem solutions. The Matrix Multiply example represents a fairly straightforward problem solution insofar as the parallel paths behave independently of one another. In other words, once the parallelisms are known, the paths can be spawned and joined together with each path contributing its part of the final solution without knowledge of what the other paths have computed. Examples of problems where there are computed, intermediate results that need broadcasting have also been explored using the *SequenceL* interpreter.

For example, the Forward Processing in the Gaussian Elimination Solution of systems of linear equations has been executed with good results in terms of finding inherent parallelisms. With three or more equations, intermediate results must be known to all paths in order to produce the final result.

Finally, in terms of scheduling, both the matrix multiply and the Gaussian Codes are examples of problems for which static a-priori schedules can be generated. The paths of execution can be determined based upon the dimensions of the matrix, in the matrix multiply, and the number of equations, in the Gaussian code. The Quicksort problem was run as an example of a problem where dynamic scheduling is necessary.

The classes of problems represented by the Gaussian and Quicksort problems already explored in the interpreter will indeed introduce additional issues in the development of the *SequenceL* compiler and scheduler.

REFERENCES

[1] Chandra R et al, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, San Diego CA., 2001.

[2] Daniel E. Cooke, "An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience,* Vol. 26, No. 11, November 1996, 1205-1246.

[3] Daniel E. Cooke, "SequenceL Provides a Different way to View Programming," *Computer Languages* 24 (1998) 1-32.

[4] Daniel E. Cooke and Per Andersen, "Automatic Parallel Control Structures in SequenceL," *Software Practice and Experience*, Volume 30, Issue 14, (November 2000), 1541-1570.

[5] Lam / MPI Parallel Computing http://www.mpi.nd.edu/lam/ 2000.

[6] Laudon J. and Lenoski D., "The SGI Origin: A ccNUMA Highly Scalable Server," *Silicon Graphics, Inc.* Mountain View, CA., http://www.sgi.com, 1999.

[7] Luecke G. R. and Lin W., "Scalability and Performance of OpenMP and MPI on a 128-Processor SGI Origin 2000", *Iowa State University*, August 16 2000.

[8] http://webten.damien.edu/mhpcc_old/WW216.html, 2000.

[9] Narlikar G. J. and Blelloch G. E., "Pthreads for Dynamic and Irregular Parallelism," *Proceedings of SC98: High Performance Networking and Computing*, Nov 1998.

[10] Pizzi J, A Master's Thesis: A SequenceL Compiler, *Texas Tech University*, 2001.

Extending FLAVERS to Check Properties on Infinite Executions of Concurrent Software Systems

Gleb Naumovich
Polytechnic University, Brooklyn
Department of Computer and Info Science
Brooklyn, NY 11201
(718) 260-3554
gleb@poly.edu

Lori A. Clarke
Computer Science Department
University of Massachusetts
Amherst, MA 01003
(413) 545-2013
clarke@cs.umass.edu

ABSTRACT

In this paper, we extend the data flow based finite state verification approach used in FLAVERS so it is applicable to infinite as well as finite executions. We first describe the previously developed algorithm that is used to check properties specified in regular languages on finite executions of distributed systems. We then present two new algorithms that enable FLAVERS to check safety or liveness properties on infinite executions of distributed software systems.

The attractiveness of FLAVERS is in its low-order polynomial complexity bounds, its ability to derive the model of executable behavior automatically from a program's source code, and its ability to improve the precision of the analysis by incrementally improving the accuracy of the program model. All these features are preserved in the proposed extensions. These extensions apply to the existing FLAVERS prototypes for analysis of both Ada and Java programs.

KEYWORDS

Finite state verification, model checking, static analysis, safety, liveness.

1 INTRODUCTION

Finite state verification techniques can be used for detecting the presence or proving the absence of certain kinds of errors in software systems. These approaches are based on reasoning about a finite, abstracted model of a system's behaviors. FLAVERS (FLow Analysis for VERification of Systems) is a finite state verification approach that uses data flow analysis techniques to verify user-specified properties of sequential and concurrent software systems [11, 12]. FLAVERS is capable of verifying properties about sequences of events, where the events are recognizable actions in the program and the sequences are either translated into or specified directly as a finite state automaton (FSA). The attractiveness of this approach is in its low-order polynomial complexity bounds, its ability to derive the model of executable behavior automatically from a program's source code, and its ability to improve the precision of the analysis by incrementally improving the accuracy of the program model. FLAVERS prototypes have been developed for Ada [11] and Java [16].

To date, FLAVERS has been restricted to considering programs with only finite executions. This is a serious limitation, because, in practice, distributed systems are frequently intended to execute infinitely. In this paper we propose two extensions to the FLAVERS analysis algorithm that allow FLAVERS to check properties on software systems with infinite executions.

Traditionally, verification properties have been classified into two broad categories: safety and liveness [1,3]. The distinction is that safety properties are finitely refutable and liveness properties are never finitely refutable. Intuitively, a safety property specifies that an undesirable state of the system is never reached and a liveness property specifies that a desirable state of the system is eventually reached on all executions.

Any property can be represented as a union of a safety property and a liveness property [2]. Any property checked only on finite executions of a system is a safety property, since the so-called "undesirable" state of the system can be viewed as the terminal state[1] where the predicate of interest either holds or doesn't hold. Safety properties are also a concern for infinite executions, when the property is finitely refutable. Thus, a property that can be proved by examining all finite execution trace prefixes would be such an infinite safety property. In this paper we focus on extending FLAVERS to handle *both safety and liveness* properties for *infinite* executions.

[1] Without loss of generality, we can assume that there is a single terminal state.

Our extension for checking safety properties on infinite executions is very simple and requires two small modifications to the original FLAVERS' approach: (1) a modification of the property representation and (2) a simple change in the analysis algorithm, where property violations are checked not only in the terminal state of the system but also at relevant intermediate points of system execution.

Our extension for checking liveness properties with FLAVERS is based on representing the property of interest as a Büchi automaton [22] and computing the states that this automaton can be in at different points of the program execution. This computation is carried out by the same data flow algorithm that FLAVERS uses for checking properties on finite executions. After that we determine if the graph contains infinite paths with suffixes on which the property Büchi automaton never enters an accept state. An existence of such a path signifies that the property represented by the automaton does not hold on the execution of the system corresponding to this path through the graph. Checking this is based on computing maximal strongly-connected components in the derived representation of the program. This algorithm could also be applied to safety properties for infinite executions, but its worst-case bound is larger than that of the algorithm specialized for safety properties on infinite executions.

Our algorithm for checking safety properties on infinite executions uses a form of FSA-based property specification, used, for example, in [7] and makes a fairly obvious change to the original data flow algorithm of FLAVERS. Our algorithm for checking liveness properties is similar to the existing algorithms used by model checking [9] and reachability analysis [5, 13] approaches. Despite these similarities, we make several important contributions. First, our proposed extensions maintain the current strong points of FLAVERS, using an efficient data flow algorithm, automatically dealing with software systems at the implementation level (although FLAVERS can handle high-level specifications as well [17]), and giving the analyst the opportunity to improve the precision of the analysis incrementally by deferring the modeling of certain features of the system until it becomes clear that such modeling is necessary. Second, we can use the existing FLAVERS framework for specifying fairness or other conditions that should be assumed during infinite executions of the system. Finally, we do not assume that all loops in the threads of control of the system can execute infinitely. Instead, the analyst has the means of specifying which of the thread loops can or cannot execute infinitely.

For convenience, we introduce the following abbreviations. We will refer to the original algorithm of FLAVERS [11] as *finite executions*, or *FE*, algorithm; to the proposed algorithm for checking safety properties on infinite executions as *safety infinite executions*, or *SIE*, algorithm; and to the proposed algorithm for checking liveness properties as *liveness infinite executions*, or *LIE*, algorithm.

In the next section we give a brief overview of the existing techniques for checking liveness properties on infinite executions of software systems. In Section 3 we describe the *FE* algorithm. Section 4 describes the specification of safety properties on infinite executions and introduces the *SIE* algorithm. In Section 5 we describe the specification of liveness properties and introduce the *LIE* algorithm. In Section 6 we discuss some the issues related to fairness conditions and incremental precision improvements. Finally, in Section 7, we outline directions for future work.

2 RELATED WORK

There exists a considerable amount of work on finite state verification approaches for verifying concurrent systems. There are four major approaches to finite state verification: reachability analysis, necessary conditions, model checking, and data flow analysis approaches. In this section we describe the way in which some representative finite state verification techniques handle properties on infinite executions.

SPIN [13] is a reachability analysis technique that accepts properties expressed in linear temporal logic (LTL) and focuses on systems with asynchronous concurrency control. Each of the threads of control in the software system is modeled with a Büchi automaton and the negation of a property is also represented as a Büchi automaton. All these Büchi automata are combined in a synchronous cross-product, with the worst-case size of this automaton being exponential in the number of threads of control. If the language of the resulting Büchi automaton is non-empty, it means that the property can be violated. This can be determined in time linear in the number of states and transitions in the combined Büchi automaton by performing the Tarjan depth-first search algorithm [20] for constructing all strongly-connected components. If there exists a reachable strongly-connected component that contains at least one accepting state, a reachable acceptance cycle exists, and so the property violation is found. The complexity of SPIN analysis is $\mathcal{O}(S + V)$, where S is the number of states in the product Büchi automaton and V is the number of transitions in this automaton.

Enhanced Compositional Reachability Analysis (ECRA) [8] works similarly to SPIN. ECRA computes a cross product of automata in a compositional way, "hiding" some of the transitions and thus, potentially, reducing the size of the cross product automaton. For safety properties, ECRA uses FSAs to represent all threads of control as well as to represent the property, which is augmented with a special trap state that represents property violations. After the FSAs for the property and threads of control are composed into a single cross-product automaton, the property is violated if this cross-product automaton contains a trap state. The analysis of liveness properties with ECRA is done in a similar way [5], using Büchi automata instead of FSAs. The property is considered to be violated if there is a reachable strongly-connected component in the product Büchi automaton that does not contain transitions to accepting states of this automaton. Conversely, if each reachable strongly-connected component contains at least one transition into an accepting state of the product automaton, the property holds. The latter holds true only under a relatively strong fairness assumption that each transition in a reachable strongly-connected component is eventually executed if this strongly-connected component is executed forever (the semantics of the distributed systems for which ECRA is designed make this assumption possible). The worst-case complexity of ECRA is the same as that of SPIN, but a good selection of the decomposition of the system model may result in significant reductions in the size of the product automaton in practice.

Necessary conditions analysis [10] generates a set of integer linear inequalities that represents necessary conditions for the existence of legal executions for systems with synchronous concurrency control. The necessary conditions express constraints on the number of times certain system events take place relative to other system events. The negation of the property is also represented as a set of inequalities. For infinite executions, some of the constraints are computed using strongly-connected components of Büchi automata, where each automaton represents a thread of control in the system. For liveness properties, some additional inequalities have to be introduced. Integer linear programming is used to solve this set of inequalities. If no solution of the set of inequalities exists, the property holds on all executions of the system. This approach is NP-hard in the size of the system of integer inequalities that has to be solved. In practice, this approach is often very efficient, although it does not appear to be applicable to asynchronous communication mechanisms.

Model checking [9] does not make a clear distinction between safety and liveness properties. In this approach, it is assumed that all executions of the system are infinite and properties are represented in computation tree logic (CTL). To prove a CTL formula F, model checking constructs a Kripke structure [14] for the system. This Kripke structure represents the set of all reachable states of the system and thus the number of its states is exponential in the number of threads of control and modeled variables. The goal of model checking is to check whether or not formula F holds in the start state of the Kripke structure, which signifies that it holds for all possible executions of the system.

In general, the complexity of model checking is $\mathcal{O}((V + E)f)$, where f is the size of the CTL formula representing the property, V is the number of states, and E is the number of transitions in the Kripke structure. The algorithm for checking liveness properties with FLAVERS that we propose in this paper is quite similar to this specific case of model checking.

3 THE *FE* VERSION OF FLAVERS

In this section we introduce the FSA-based property specification used by FLAVERS, give a very high-level overview of FLAVERS, and then present the *FE* algorithm.

3.1 Representing *FE* Properties

FLAVERS uses an event-based view of the software system being analyzed. In this view, user-selected names, called *events*, are associated with observable activities of interest in the system and then all potential executions of the system are represented as sequences of these events. For example, both a variable assignment and a method call could be examples of events.

A number of formalisms for specifying properties have been proposed, including temporal logics [9,19], process algebras [4, 15], and various forms of regular languages and finite state automata [18,21]. FLAVERS uses deterministic finite state automata for specifying properties to be checked on terminating executions of a system.

A deterministic FSA can be represented as a tuple $(S, s_0, \Sigma, \delta, A)$. S is the set of all *states* of the FSA, including the unique start state s_0. Σ is called the *alphabet* of the FSA and includes all events used by this FSA. δ is a total *transition function* $S \times \Sigma \to S$ that represents all event-based transitions between the states of the FSA. We deal with *total* FSAs, which means that from any state there is a transition based on each event from the alphabet[2]. We write $\delta(s, e) = s'$ to indicate that there is a transition from state s to state s' based on event e. Finally, A is the set of *accept* states $\{a_1, a_2, ..., a_p\}$, $\forall 1 \leq i \leq p, a_i \in S$. A *trace* of an FSA on an event sequence $w = e_1, e_2, ..., e_n$ is a sequence of states $s_0, s_1, ..., s_n$, where s_0 is the start state and for any $i, 1 \leq i \leq n$, there is a transition from s_{i-1} to s_i on event e_i. A sequence of events $e_1, ..., e_n$ is *accepted* by P if the last state in the corresponding trace of this automaton is an accept state: $s_n \in A$. An example of an FSA is given in Figure 1. This FSA has two states s_0 and s_1, and so $S = \{s_0, s_1\}$. State s_0 is the start state, which is denoted by an arrow with no origin, and also an accept state, as denoted by concentric circles. The alphabet of this FSA is $\{$open, close, C$\}$. There is a transition from state s_0 to state s_1 based on event open. Graphically, we may represent several transitions from state s to state s' with a single arrow that is labeled with a list of events on which all these transitions are based. For example, in Figure 1 the self-arrow on state s_1 is labeled open, C and thus represents two transitions, $\delta(s_1, \text{open}) = s_1$ and $\delta(s_1, \text{C}) = s_1$. This FSA accepts the sequence open, C, close, because it has a trace s_0, s_1, s_1, s_0 on this sequence and the last state in this sequence, s_0, is an accept state.

We call the set of properties that can be specified as an FSA *regular event sequencing* properties. Such a property holds for a system if for any terminating execution of this system the sequence of events observed on this execution puts the FSA in an accept state.

[2] For convenience, we introduce a single trap state that is the destination node of all "illegal" transitions introduced to make the FSA total.

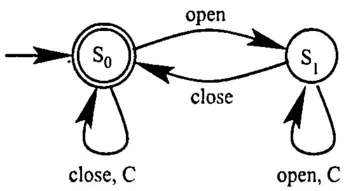


Figure 1: An example FSA or Büchi automaton

3.2 Overview of the *FE* Approach

FLAVERS models the software system under analysis as a *Trace Flow Graph (TFG)*. The TFG is based on the control flow graphs (CFGs) for the components of the system, where the nodes in the TFG may be labeled with events. We call the collection of all events with which the nodes of the TFG are labeled the *alphabet* of this TFG. To reduce the size of the representation, the CFGs are refined to remove all nodes that are not labeled with an event or that do not affect the sequencing of events. Thus, the resulting refined CFGs correctly capture all possible sequences of events associated with their corresponding component. At present, FLAVERS handles interprocedural systems by in-lining called routines. Since nodes with events are usually a small subset of all the nodes in the original CFG, the refined CFG is typically much smaller than the original CFG. Thus, in our experience, in-lining of refined CFGs usually does not cause a severe blow-up in the size of the CFG representation.

The TFG for a concurrent system is obtained by connecting the refined, in-lined CFGs for all threads of control with additional nodes and edges. Unique *initial* and *final* nodes represent the start and the end states of the system respectively. In addition, depending on the concurrency semantics of the system being modeled, the TFG may include special nodes that represent communication among the threads of control. In all cases, special edges that represent interleavings of events from the threads of control executing in parallel are added to the TFG. Each path from the initial to the final node in the TFG represents a sequence of events that occur on the nodes along this path. The TFG is a *conservative* representation of the sequences of events that could occur along a system execution. That is, any sequence of events in the TFG alphabet that could occur during execution of the system is represented by some path in the TFG with a corresponding event sequence. However, the converse is not true, since CFGs and thus TFGs may contain a number of *infeasible paths*, which do not correspond to any system executions.

Formally, a TFG is a labeled directed graph $G = (N, E, n_{initial}, n_{final}, \Sigma_G, L)$, where N is the set of graph nodes, E is the set of edges, $n_{initial} \in N, n_{final} \in N$ are the initial and final nodes, Σ_G is an alphabet of event labels associated with the graph, and $L : N \to \Sigma_G$ is a function that labels some of the nodes of the graph with an event drawn from this alphabet. For convenience in presenting the algorithms, with each node n in the TFG, we associate a set $\mathit{Pred}(n)$ containing all predecessors of n.

A property specified as an FSA holds for a system if this FSA accepts event sequences for all paths through the TFG for this system. FLAVERS uses the data flow based *FE* analysis algorithm to solve this problem. This is done by associating states of the property FSA with the nodes of the TFG. We use a forward-flow data flow algorithm where states are propagated from one node to another, depending on the FSA transition function associated with the events that are encountered in the TFG. Thus, a state s is associated with node n if and only if there is a path from the initial node of the TFG to n that encounters a sequence of events that drives the property FSA to state s when the path reaches n. Note that since multiple paths may exist from the initial node to node n, a set of property states may be associated with each node. The iterative worklist algorithm continues to propagate states to nodes in the TFG until it reaches a fixed point, where no additional states can be associated with TFG nodes. The outcomes of this analysis are either that (1) the set annotating the final node of the TFG contains only accept states of the FSA, indicating that the property holds on **all** executions of the system or (2) the set annotating the final node of the TFG contains at least one non-accept state of the FSA, which means that the property **may** not hold on **some** executions.

The alphabet of the property must be a subset of the events in the alphabet of the TFG: $\Sigma \subset \Sigma_G$. To represent the fact that the property "ignores" the events not in Σ, we can modify the transition function of the FSA to contain self-transitions on all states of the property for all events that are in the TFG alphabet but not in the FSA alphabet: $\forall s \in S, \forall e \in (\Sigma_G \setminus \Sigma), \delta(s, e) = s$. As a result of this modification, the alphabet of the property becomes equal to the alphabet of the TFG: $\Sigma = \Sigma_G$.

If the analysis finds that a property holds on all paths through the TFG, then it is guaranteed to hold on all possible executions of the system. When the analysis indicates that the property does not hold on some paths through the TFG, this may be because the system is in error or it may be because all the paths in the system model that violate this property correspond to infeasible paths. FLAVERS provides a means for selectively removing infeasible paths from consideration by allowing the analyst to add *feasibility constraints*, finite state automata that model semantic restrictions on the system's execution that are not reflected in the TFG. For example, CFGs, and the TFGs constructed from them, typically do not model the values assigned to variables during execution. Thus, paths through the TFG may not represent feasible executions because these paths do not respect the values of some variables. A feasibility constraint could be constructed to track the possible finite values or ranges of values of such a variable, thereby eliminating some or all infeasible paths. Formally, a constraint automaton is an FSA $C = (S_C, s_C, \Sigma_C, \delta_C, c_C)$, where c_C is a unique *crash* state.

The crash state of a feasibility constraint signifies that the sequence of events applied to the constraint does not correspond to any legal behavior of the system. For any state $t \in S_C$ and any event $e \in \Sigma_C$, $\delta_C(t,e) = c_C$ if and only if observing event e at state t does not correspond to any legal behavior of the constraint. The crash state is a sink, which means that there are no transitions from this state to any other state in the constraint. When feasibility constraints are used, instead of propagating states of the property automaton through the TFG, the *FE* algorithm propagates tuples of states where each tuple has an element that represents a state of the property and an element for a state of each of the constraints. More precisely, *tuple* $T = (p, c_1, ..., c_k)$, where $p \in S_P, c_i \in S_{C_i}, \forall 1 \leq i \leq k$ and the *start* tuple T_0 is the tuple $(s_P, s_{C_1}, ..., s_{C_k})$. If one of the elements in a tuple represents a crash state for a constraint, this tuple is not propagated beyond this node.

Similar to the case where no feasibility constraints are used, the *FE* algorithm runs until it reaches a fixed point, after which the states of the property annotating the final TFG node determine whether the property holds on all executions of the system.

In the following, we refer to the collection consisting of the TFG, property automaton P, and the constraint automata $C_1, ..., C_k$ as an *analysis problem*. Similar to extending the alphabet of the property, we extend the alphabets of all constraints to include all events in the alphabet of the TFG: $\forall i, 1 \leq i \leq k, \forall e \in (\Sigma_G \setminus \Sigma_{C_i}), \forall s \in S_{C_i}, \delta(s,e) = s$.

We refer to the collection of all possible tuples for a given analysis problem as *Tuples*:

$$\mathit{Tuples} = \bigcup_{p \in S_P} \bigcup_{c_1 \in S_{C_1}} \cdots \bigcup_{c_k \in S_{C_k}} (p, c_1, ..., c_k)$$

A *tuple transition function* $\Delta : N \times \mathit{Tuples} \rightarrow \mathit{Tuples}$ describes propagation of tuples through TFG nodes. It is defined as follows:

$$\forall n \in N, T = (p, c_1, ..., c_k) \in \mathit{Tuples}, \Delta(n,T) = (\delta_P(p, L(n)), \delta_{C_1}(c_1, L(n)), ..., \delta_{C_k}(c_k, L(n)))$$

3.3 The *FE* Algorithm

The *FE* algorithm of FLAVERS is a forward flow data flow algorithm over the TFG with the power-set of *Tuples* as the lattice. The function space is provided by function $\Omega : 2^{\mathit{Tuples}} \times N \rightarrow 2^{\mathit{Tuples}}$ based on the tuple transition function Δ[3]:

$$\forall n \in N, A \in 2^{\mathit{Tuples}}, \Omega(A,n) = \{T | \exists T' \in A, \Delta(T', n) = T\}$$

The *FE* algorithm associates two sets of tuples with each node n in the TFG, $\mathit{IN}(n)$ and $\mathit{OUT}(n)$. The *IN* set for node n represents the possible states of the system immediately before this node is executed. This set is computed as the union of all possible states in which the system can be after the predecessor nodes for n are executed:

$$\mathit{IN}(n) = \bigcup_{p \in \mathit{Pred}(n)} \mathit{OUT}(p)$$

The *OUT* set for node n represents the possible states of the system immediately after this node is executed. This set is computed by applying the transition function to n and the tuples in its *IN* set and removing from the result all tuples that contain at least one constraint crash state:

$$\mathit{OUT}(n) = (\bigcup_{T \in \mathit{IN}(n)} \Delta(n,T)) \setminus \{T = (p, c_1, ..., c_k) \in \mathit{Tuples} | \exists i, 1 \leq i \leq k, c_i = c_{C_i}\}$$

The algorithm is initialized by setting the *OUT* set of the initial TFG node to contain the start tuple and setting all other *IN* and *OUT* sets to be empty.

The algorithm repeatedly recomputes *IN* and *OUT* sets of the TFG nodes in an arbitrary order, until a fixed point is reached. To determine if the property holds on all terminal executions of the system, all tuples in the *OUT* set of the final TFG node are investigated. The property holds if all states of the property FSA in these tuples are accept states: $\forall T = (p, c_1, ..., c_k) \in \mathit{OUT}(n_{final}) : p \in A_P$. If this condition is not true, FLAVERS concludes that the property does not hold.

4 THE *SIE* VERSION OF FLAVERS

By making a simple modification to the *FE* algorithm, FLAVERS can check safety properties on infinite executions. The most important change is in the representation of properties. Although we still use FSAs to represent safety properties to be checked on infinite executions, these FSAs have a somewhat different form than those used in the *FE* algorithm.

[3] To satisfy the definition of a function space, several additional functions also need to be defined, but they are not important for the discussion in this paper. An interested reader is referred to [12].

Any event sequencing safety property can be formulated in a form that describes undesirable behaviors of the software system under analysis. The reason for this is that safety properties are finitely refutable statements and so they can be represented as sequences of events that should be observed to refute the property. The refutation event must be explicit and thus represents a certain point in the event sequences. Similar to the approach of [6], we define a special *violation* state v, which represents the property being refuted. v is a sink state, which means that there is a transition from v to v on any event in the alphabet of this FSA. We say that sequences of events that correspond to traces of this FSA that contain the violation state v *violate* the safety property represented by this FSA. The following theorem offers a proof that any regular event sequencing safety property can be represented as an FSA with a violation state.

Theorem 1. *Any regular event sequencing safety property can be represented as an FSA with a unique violation state v, such that the property does not hold on an event sequence if and only if the trace of the FSA corresponding to this sequence contains v. The converse is true as well: any FSA with a violation state represents a safety property.*

Proof. Due to the space limitations, we do not present the full formal proof, which is based on the formal definition of safety [1]. □

The *SIE* algorithm of FLAVERS uses FSAs with a violation state to represent properties. This algorithm proceeds in exactly the same way as the *FE* algorithm, with the exception that instead of checking just the final node of the TFG for violations, we check all nodes. It is not sufficient to check only the final node, because it represents the terminal state of the system, in which all threads of control terminated. This terminal state is never reached if at least one thread enters an infinite loop. Thus, the *SIE* algorithm checks if any node n of the TFG contains a tuple T such that the property in this tuple is in the violation state; this represents a violation of the property.

The worst-case complexity of the *SIE* algorithm is the same as that of the *FE* algorithm, $O(N^2S)$, where N is the number of nodes in the model of the software system under analysis and S is the number of states in the synchronous cross-product of the automaton representing the property of interest and all feasibility constraint automata used by FLAVERS to improve its analysis precision.

5 THE *LIE* VERSION OF FLAVERS

In this section we first describe the representation of liveness properties used in our *LIE* FLAVERS algorithm, present an overview of the approach, and then give the details of the *LIE* algorithm itself.

5.1 Representing Liveness Properties

Since FSAs can encode only finite event sequences, we need a different formalism to describe infinite behaviors. ω-automata [21] provide such a formalism. Usually, for an infinite trace sequence σ to be accepted by an ω-automaton, some infinite pattern of accept states of this automaton must be observed on the traces of this automaton on σ. In particular, we use a well-known subclass of ω-automata, Büchi automata. A *deterministic* Büchi automaton is an automaton $(S_B, s_B, \Sigma_B, \delta_B, A_B)$, where S_B is a set of states, $s_B \in S_B$ is a start state, Σ_B is the alphabet, A_B is a set of accept states, and δ_B is the transition function $S_B \times \Sigma_B \to S_B$. A trace of a Büchi automaton on an infinite event sequence $\sigma = e_1, e_2, \ldots$ is an infinite sequence of states $s_0, s_1, \ldots$, where s_0 is the start state and for any $i \geq 1$, there is a transition from s_{i-1} to s_i on event e_i. A Büchi automaton accepts an infinite sequence of events $e_1, e_2, \ldots$ if the corresponding trace contains an infinite number of accept states. For example, the automaton in Figure 1 can be interpreted as a Büchi automaton. An infinite event sequence of alternating `open` and `close` events `open`, `close`, `open`, `close`, ... is accepted by this automaton because the corresponding trace of this automaton $s_0, s_1, s_0, \ldots$ contains an infinite number of occurrences of accept state s_0.

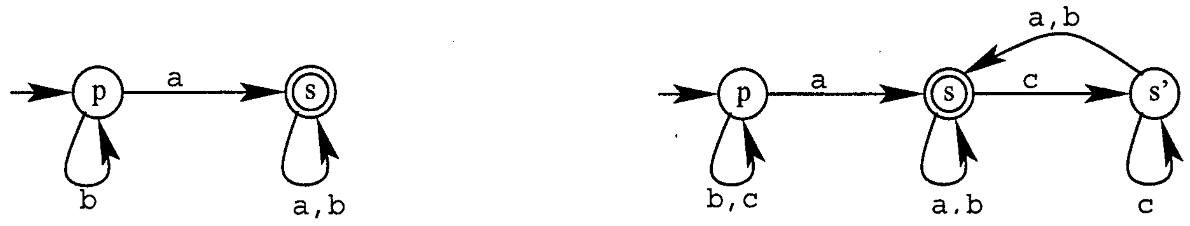
An arbitrary Büchi automaton cannot be used as a liveness property in our approach. The reason for this is that in FLAVERS, if the event associated with a node is not in the alphabet of a (property or constraint) automaton, the automaton does not change state when the transition function for tuples is used to compute the *OUT* set for this node. Thus, it is possible that an execution trace has a suffix in which all tuples have the Büchi property automaton in its accept state because none of the events in this trace is in the alphabet of this property. To avoid this situation, we modify each Büchi property automaton in a way that makes its alphabet equal to the alphabet of the TFG. The modification is based on creating an additional non-accept state for each accept state in the Büchi automaton and having transitions on events that are not in the alphabet of the Büchi automaton go from each accept state to its newly created non-accept state.

For this modification, let A be a Büchi automaton for which $\Sigma_A \subset \Sigma_T$, where Σ_T is the alphabet of the TFG. We build a new Büchi automaton A' equivalent to A in the sense that it accepts the same set of infinite strings in the following way:

1. Copy A to A'.

2. Set $\Sigma_{A'} = \Sigma_T$.

3. For each non-accepting state $s \in S_{A'}$, create new transitions $\delta_{A'}(s, e) = s$ for each $e \in \Sigma_T \setminus \Sigma_A$.



(a) An example property before transformation

(b) An example property after transformation

Figure 2: Illustration for the algorithm transforming Büchi automata

4. For each accepting state $s \in S_{A'}$, create a new non-accepting state $s' \in S_{A'}$. For each transition originating in s, $\delta(s,e) = s''$, create a transition $\delta(s',e) = s''$. Note that this includes cases where $s'' = s$. Create a transition from s to s' for each event in $\Sigma_T \setminus \Sigma_A$. Finally, create a self-transition on state s' for each event in $\Sigma_T \setminus \Sigma_A$: $\delta(s',e) = s'$.

Note that after this modification, all transitions based on events that were not originally in the alphabet of the property that originate in an accepting state s lead to a new state s' created by the algorithm. Figure 2 illustrates the Büchi automaton transformation defined here. We assume that $\Sigma_T = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$ and $\Sigma_A = \{\mathtt{a},\mathtt{b}\}$. Figure 2(a) shows a Büchi automaton before the transformation and Figure 2(b) shows the corresponding Büchi automaton after the transformation. Note that this transformation does not cause a severe blow-up in the size of the Büchi property automaton, since the number of extra states created equals the number of accept states in the original Büchi property automaton.

5.2 Overview of the *LIE* Approach

A direct and very naive approach to checking properties on infinite executions would be to follow the *FE* algorithm with a Büchi automaton representing the property of interest, but to preserve the history of changes for each (T,n) pair, starting with the initial pair $(T_0, n_{initial})$. Then we can check if the current state is already present in this history and if an accept state of the property has been entered since its last occurrence. Of course, the complexity of storing and perusing all that additional history information is prohibitive. Instead, we use the *FE* algorithm but then evaluate the TFG with all the tuples assigned to its nodes to find infinite behaviors. In the rest of this section we give the details of this *LIE* algorithm and the artifacts that it relies upon.

As described in Section 3.3, the *FE* algorithm of FLAVERS associates sets of tuples $OUT(n)$ with each node n in the TFG. A tuple T is in $OUT(n)$ if there is a path through the TFG from the initial node $n_{initial}$ to n that corresponds to a trace of events that would cause the automata for the property and all constraints to transfer from their start states to the states represented by tuple T. Thus, the problem of determining whether a particular tuple T appears in the *OUT* set of node n can be viewed as a reachability problem in the *tuple-node* space *Tuples* × N. Formally, the tuple-node space *Tuples* × N is a structure (P, E_{tn}), where P is the set of pairs (T,n) such that $T \in OUT(n)$ after the *FE* state algorithm terminates and E_{tn} is the set of edges, where $((T_1,n_1),(T_2,n_2)) \in E_{tn}$ if $(T_1,n_1),(T_2,n_2) \in P \wedge n_1 \in Pred(n_2) \wedge \Delta(n_1,T_1) = T_2$.

We say that there exists a *path* from pair (T,n) to pair (T',n') if there are pairs $(T_1,n_1),...,(T_k,n_k)$ for some $k \geq 0$, such that $((T,n),(T_1,n_1)),((T_1,n_1),(T_2,n_2)),...,((T_k,n_k),(T',n')) \in E_{tn}$. A *reachability function Reach* : $P \to 2^P$ for a given pair returns the set of all pairs that can be reached for this pair through a path in the tuple-node space: $\forall (T,n) \in$ *Tuples* $\times N, Reach((T,n)) = \{(T',n') | \exists$ a path from (T,n) to $(T',n')\}$.

From an abstract level, our *LIE* algorithm uses an approach for analyzing the tuple-node space that is very similar to the approach used by model checking and reachability analysis approaches. We attempt to identify strongly-connected components in the tuple-node space that do not have tuples containing an accept state of the property. If such a strongly-connected component is found, it represents one or more infinite executions on which an accept state of the property is not entered infinitely often. By the definition of Büchi automata acceptance, the property is violated on such executions. On the other hand, the absence of such strongly-connected components signifies that the liveness property holds on all infinite executions of the program. In the rest of this section we describe this algorithm in detail.

5.3 The *LIE* Algorithm

The following algorithm for checking liveness properties with FLAVERS assumes that the *FE* algorithm is used first, and so every node of the TFG has a set of tuples, denoted *OUT*, associated with it. The following steps are then performed:

1. Remove from the *OUT* sets of all TFG nodes all tuples where the Büchi automaton is in an accept state.

2. Find all maximal strongly-connected components in the resulting (reduced) tuple-node space. A maximal strongly-connected component in the tuple-node space $Tuples \times N$ is defined as a set of tuple-node pairs $C \subseteq P$ such that
   (a) $\forall (T_1, n_1), (T_2, n_2) \in C, (T_2, n_2) \in Reach((T_1, n_1))$ and
   (b) $\forall (T_1, n_1) \in C, (T_2, n_2) \in P \setminus C, (T_1, n_1) \notin Reach((T_2, n_2)) \vee (T_2, n_2) \notin Reach((T_1, n_1))$.

3. If at least one strongly connected component has been found, the property is violated. This property violation can be illustrated by inserting in the *OUT* sets of all TFG nodes the tuples that were removed in step 1 of this algorithm and showing a path from $(T_{initial}, n_{initial})$ to this strongly-connected component.

Intuitively, if after removing all tuples in which the property automaton is in an accept state, no strongly-connected components exist in the tuple-node space, it means that no execution can be found on which the property automaton enters an accept state only a finite number of times. This means that the liveness property being checked holds on all possible program executions. Alternatively, if a strongly connected component is found, it represents a suffix of an infinite execution such that on this suffix no accept states of the property are entered. Thus, on this execution an accept state of the property is entered only a finite number of times, and so the property is violated.

This approach is similar to the one used by model checking [9]. In fact, our approach can be reduced to checking a specific CTL formula $AGAFa$ with model checking, where a is true in a tuple-node pair (T, n) if and only if the state of A in T is accepting. The major difference between our approach and that of model checking is in the way that the state space of the system is represented and in the way this representation is computed.

5.4 Properties of the *LIE* Algorithm

We need to prove termination and conservativeness, and determine the complexity of this algorithm.

Theorem 2 (Termination). *For any LIE analysis problem $(G, P, C_1, ..., C_k)$, the algorithm terminates.*

Proof. This follows from the fact that the tuple-node space is finite and termination of the efficient Tarjan algorithm [20] for computing maximal strongly-connected components. □

Conservativeness of our algorithm means that if there exists an execution of the system on which the Büchi property automaton does not hold, the algorithm will detect that.

Theorem 3 (Conservativeness). *If there exists an execution of the system on which there is a suffix where an accept state of the property Büchi automaton is not reached infinitely often, our algorithm will detect that.*

Proof. Suppose that there is an execution of the system with a suffix on which the Büchi automaton never enters an accept state. Since both the TFG model and the *FE* algorithm are conservative [11], this means that there is a trace through the tuple-node space of the problem on which the Büchi automaton never enters an accept state. Since the tuple-node space is finite, this trace must correspond to a loop L in the tuple-node space. When our algorithm eliminates all states of the tuple-node space that correspond to tuples in which the Büchi property automaton in an accept state, loop L is still present, since in no tuples along this loop is the property in an accept state. Thus, there exists a strongly-connected component that contains this loop, and so our algorithm will conclude that the property may be violated. □

The following theorem states the worst-case complexity of the algorithm.

Theorem 4 (Worst-case Complexity). *The worst-case complexity of our algorithm as described is $O(|N|^2|Tuples| + |E_{tn}|)$.*

Proof. The $|N|^2|Tuples|$ component of the complexity formula in the statement of this theorem is just the worst-case complexity of the *FE* algorithm that must be done first. The worst-case complexity of the Tarjan algorithm for finding all maximal strongly-connected components of the tuple-node space is $O(|P| + |E_{tn}|)$. By observing that $|P| \leq |N||Tuples|$, we arrive at the stated complexity. □

This worst-case result is consistent with the complexity of other finite state verification approaches on liveness properties, except for [10], where the worst-case bound in general cannot be expressed in terms of the characteristics of the property and system models.

5.5 Implementation

We have implemented the approach proposed in this paper and carried out an initial, preliminary experiment, in which we dealt with two liveness properties for a concurrent Ada producer/consumer example. In this example, multiple producer threads put items in an unbounded buffer and multiple producer threads extract items from this buffer. Our first property specifies that a consumer thread does not starve, i.e. on all infinite executions a consumer thread extracts items from the buffer an infinite number of times. This property can be violated, since the example does not guarantee fair treatment of all threads. Our implementation correctly finds an infinite execution that demonstrates starvation of a consumer thread. Our second property specifies

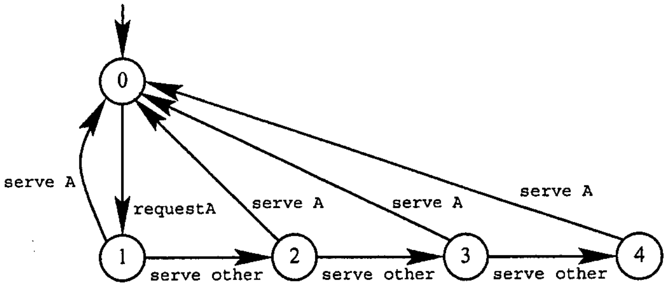


Figure 3: A fairness FSA example

that on all infinite executions some buffer activity (putting or extracting items) happens infinitely often. Our implementation correctly demonstrated that this property holds on all possible executions of the example.

The producer/consumer example is scalable; we checked the two properties described above on four different sizes of the example: 2, 4, 6, 8, where the size corresponds to the number of consumers/producers in the example. (Thus, the example of size 2 has two producers and two consumers.) For each of the sizes the outcome described in the previous paragraph was obtained. An interesting observation is that for both properties, the number of required constraints did not depend on the size of the example. For the first property we needed two constraints modeling control flow through select threads and for the second property we needed three similar constraints. In all cases, checking each of the properties took under 4 seconds on a Pentium III Xeon 550 MHz machine.

6 FAIRNESS ASSUMPTIONS AND PRECISION IMPROVEMENTS

To be conservative, FLAVERS assumes that all traces through the TFG or tuple-node space correspond to executable behavior in the system being analyzed. Constraints can be used to eliminate infeasible traces selectively. For infinite executions, the algorithm described above assumes that all loops can be executed infinitely. It would be more realistic to recognize that some loops can execute infinitely, while others cannot. Program optimization techniques could be used to statically detect at least some of the finite loops. Using FLAVERS constraint mechanism (e.g. modeling values of variables used in loop predicates), information could be provided to improve or refine this static analysis. Alternatively, we believe that it may be more practical to let the analyst mark those loops in threads that can never execute infinitely (or, the analyst may mark all potentially infinite thread loops). Given this information, the above algorithm can be modified so as not to consider the strongly-connected components in the tuple-node space that correspond to a set of loops in the control flow of individual threads, if any of these loops cannot be infinite.

Fairness conditions are often employed to ensure that some reasonable behaviors of a system are taken into account. For example, in a client-server configuration of system threads, a possible fairness requirement is that if two client threads request a service S infinitely often and the server satisfies S infinitely often, then both clients obtain the service infinitely often (in other words, it is not possible for one of the clients to "starve" while the other always gets the service). With FLAVERS, we can again use the feasibility constraint mechanism to represent fairness assumptions. Because feasibility constraints are FSAs, these assumptions are rather strong. For example, using only FSAs, it is impossible to represent the fairness assumption about the client-server system described above. However, we can represent an assumption that after client A requested service, the server can serve at most 3 requests from clients other than A before serving client A. An FSA modeling this fairness assumption is shown in Figure 3. Transitions labeled `request A` represent the event of client A requesting service and transitions labeled `serve A` and `serve other` represent the events of the server serving A and a client other than A respectively. (Note that in this example, we make two reasonable assumptions about the system: (1) a client does not post a request if it has one unsatisfied request outstanding and (2) the server does not provide an unrequested service.) We believe that such fairness conditions are practical, since they can be derived from the actual specifications of the description of the environment in which the software system under analysis has to execute, unlike fairness conditions that specify that a service will be offered infinitely often.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have extended the original data flow analysis algorithm of FLAVERS (*FE* algorithm) to check properties on infinite executions of concurrent software systems. Two different algorithms are presented, one for checking safety properties and the other for checking liveness properties. Although, by representing safety properties as Büchi automata, we could use the *LIE* algorithm for checking both kinds of properties on infinite executions, the *SIE* algorithm has better worst-case complexity bounds than the *LIE* algorithm. Both of these algorithms do not involve changing the existing analysis algorithm of FLAVERS but rather add to it, in a language independent way. This means that the feasibility constraints of FLAVERS that improve precision of the analysis can be used successfully with the proposed algorithms. This is particularly attractive since feasibility constraints can be used to model fairness assumptions about the system under analysis or to refine information about infinite and finite loops. Of course, the problem of determining precisely whether a given loop can be infinite is undecidable. Efficient, conservative automated techniques can be used for this problem and supplemented with guidance from the analyst. With such information, the precision of the analysis results would improve considerably. Thus, we believe that this approach would provide a more precise and realistic basis for analysis and incorporates application-specific fairness and executability considerations.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] G. R. Andrews. *Concurrent Programming — Principles and Practice*. Benjamin/Cummins Publishing Company Ltd., 1991.

[4] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[5] S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 227–243, Sept. 1997.

[6] S. C. Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug. 1994.

[7] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–151, Oct. 1995.

[8] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, Jan. 1999.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[10] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, Jan. 1995.

[11] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[12] M. B. Dwyer and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report 1999-52, University of Massachusetts, Amherst, Aug. 1999. `ftp://ftp.cs.umass.edu/pub/techrept/techreport/1999/UM-CS-1999-052.ps`.

[13] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[14] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, London, 1977.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, Berlin, 1980.

[16] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[17] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 77–93, Nov. 1997.

[18] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, Mar. 1990.

[19] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 46–57, Oct.–Nov. 1977.

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[21] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[22] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science*, pages 167–176, June 1987.

Qualitative modeling of hybrid systems*

Oleg Sokolsky and Hyoung Seok Hong
Department of Computer and Information Science
University of Pennsylvania
{sokolsky,hshong}@saul.cis.upenn.edu

Abstract

The paper discusses an approach to construct discrete abstractions of hybrid systems by means of qualitative reasoning. The work is performed in the context of a modeling language for hybrid systems CHARON. We introduce a qualitative version of the language and describe the abstraction technique using a motivational example. The resulting abstract model is conservative and can be used to analyze properties of the original hybrid system.

Keywords: hybrid systems, abstraction, qualitative reasoning.

1 Introduction

Distributed embedded control systems usually consist of multiple components that exhibit both continuous and discrete behavior. Hybrid systems is a widely-used mathematical model for such systems. Since many embedded systems are safety-critical, it is important to analyze hybrid systems for correctness. The combination of discrete and continuous state changes makes analysis of hybrid systems an extremely challenging task. Algorithmic verification techniques require that we work with a finite representation of the state space of a system. Abstractions and approximations are necessary to make algorithmic analysis possible. In this paper, we consider the construction of discrete approximations of hybrid systems by means of *qualitative reasoning.*

Qualitative reasoning [12, 7, 9] is a well-established technique in the Artificial Intelligence community. It allows researchers to model physical systems using incomplete information. Often, there is not enough information about the system to represent it by means of differential equations. However, the basic relations between the variables in the system are known. In this situation, qualitative models can be used to capture the incomplete knowledge in a model, which can be simulated to obtain a rough outline of the system behavior. Furthermore, as more information about the system becomes available, the qualitative model can be refined to provide a more accurate description.

An alternative role for qualitative reasoning has received much less attention. Qualitative models can be seen as discrete abstractions of continuous and hybrid systems. They provide a *conservative approximation* of the system behavior. That is, every possible behavior of a system is captured by some qualitative behavior, but not all qualitative behaviors necessarily correspond to a real system behavior. Qualitative models, which exhibit finite-state behavior, can be fully explored by a verification tool and thus provide a means of conservative analysis of hybrid systems.

We explore qualitative abstractions of hybrid systems in the context of CHARON [1], a recently introduced novel language for hybrid system modeling. The language supports specification of multi-threaded (parallel or distributed) systems as a hierarchy of concurrent agents and complex behaviors within one thread as a hierarchy of modes. CHARON has a number of high-level language features such as data encapsulation and scoping, exception handling, and instantiation of parameterized objects. CHARON has been given formal compositional semantics [2] that makes modular reasoning about hybrid systems possible. In this paper, we describe a qualitative variant of the CHARON language that will allow us to construct conservative qualitative approximations of CHARON models and analyze them using state-space exploration techniques.

*This work is supported in part by the NSF grant CCR-9988409, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA ITO MOBIES F33615-00-C-1707, and ONR N00014-97-1-0505 (MURI).

Related work. Qualitative reasoning has emerged in the past decade as a mature technique for approximate reasoning. Qualitative abstractions are primarily targeted at continuous systems expressed as differential equations. However, tools such as QSIM [12] are capable of modeling discrete transitions and are thus applicable to general hybrid systems. An application of qualitative reasoning to hybrid systems in the context of controller synthesis is discussed in [5]. Similar in spirit but different technically is recent work on verification of safety properties in continuous systems via qualitative abstractions [14, 13]. There, conservativeness of qualitative abstractions is used to prove that violations of safety properties is impossible in the concrete model. Analysis is based on reasoning about individual trajectories, while we are concentrating on more traditional in the verification are state-machine representations.

It is well-known that formal verification techniques such as reachability analysis and model checking are undecidable for hybrid systems in general [11]. Research has concentrated on decidable subclasses of hybrid systems, or on finding conservative approximations for hybrid systems. See [3] for a survey of state-of-the-art techniques.

The need to construct finite abstractions of infinite-state systems is not limited to the hybrid systems domain. Predicate abstraction [10] is a promising technique for reducing the range of a variable to a finite set of "important" values. Effectively, predicate abstraction determines appropriate landmark values for each variable in the program. The proposed approach can be seen as an extension of the predicate abstraction techniques for hybrid systems.

The paper is organized as follows: in Section 2 we introduce the language CHARON and informally describe its semantics. In Section 3, we present a framework for qualitative description of systems. Our approach follows the treatment of [12], which describes the simulation of qualitative models using a tool QSIM. Our approach is not based on simulation, however. We construct a qualitative model as a hierarchical state machine and explore its state space to determine its properties. The benefits of this approach are discussed in Section 5. Then, in Section 4 we present the qualitative variant of CHARON and its semantics. The semantics is compositional in the sense that behaviors of composite objects are computed from their components. A simple example is presented in Section 5 to illustrate the approach.

2 CHARON modeling language

CHARON is a high-level language for modular, hierarchical description of hybrid systems. CHARON describes a hybrid system as a collection of concurrent *agents* that interact with each other through shared variables and bounded-capacity channels[1]. Agents have well-defined interfaces, consisting of its input and output variables and channels. Sequential behavior is described in CHARON by means of *modes*. Modes also have interfaces, consisting of entry and exit *control points*, through which a thread of control enters and leaves the mode.

Intuitively, an execution of a CHARON specification is an alternating sequence of *discrete* and *continuous* steps. Discrete steps are instantaneous mode switches, while continuous steps take a finite amount of time when no control changes occur.

The hierarchy in CHARON is twofold. The *architectural hierarchy* describes how the *agents* in the system interact with each other, hiding the details of interaction between sub-agents. The *behavioral hierarchy* describes behavior of each agent as a collection of *modes*, hiding the low-level behavioral details. At the leaves of the architectural hierarchy are *primitive* agents that do not have concurrent sub-agents. Behaviors of primitive agents are captured by *modes*, described below.

Agents and modes operate on sets of typed variables. In each agent or mode, variables are partitioned into global and local variables. Global variables are further categorized into input and output variables. Also, variables can be either analog or discrete. Discrete variables are updated by discrete steps during the execution; analog variables are updated in a continuous fashion, but may also be reset by discrete steps. During a continuous step, analog variables follow a *flow*, a smooth continuous function of time. We assume that analog variables have type real.

A mode is a hierarchical hybrid state machine equipped with analog and discrete variables. While a mode stays in a state, its analog variables are updated continuously according to a set of constraints, which take the form of differential and algebraic equalities and inequalities. Taking transitions from one state to another, the mode updates its discrete variables. States of the mode are submodes that can have their own behavior. A mode has a number of control points, through which control enters and exits the mode. That is, to perform a computation in one of its submodes, a mode takes a transition to an entry point of that submode. When the computation in the submode is complete, a transition from an exit point of the submode is taken. The mode also has entry transitions, from

[1] Channels are not considered in this paper.

an entry point of the mode to an entry point of one of its submodes, and exit transitions, from an exit point of a submode to an exit point of the mode. Entry transitions specify initial states of a mode and may give initial values to the variables of the mode.

Primitive modes, which do not have any submodes, can have multiple entry points but only the default exit point. Since there are no internal control points in a primitive mode, every entry transition is also an exit transition. Intuitively, a primitive mode stays during its execution in its default exit point.

Transitions are labeled with *guards* and *actions*. A guard is a predicate on the values of the mode variables. A transition is *enabled* when its guard is true. An action is a partial *state transformer*: when a transition is taken, variables of the mode are updated according to the action of the transition.

Before the computation of a mode is completed, it may be interrupted by a group transition, originating from a default exit point of the mode. After an interrupt, control is restored to the mode via a default entry point. We use *invariants* to force one of the outgoing transitions. Control can reside in a mode only as long as its invariant is satisfied. As soon as an invariant is violated, control has to leave the mode by taking one of the enabled outgoing transitions.

Each primitive agent has an associated *top-level* mode that specifies its behavior. A top-level mode has a single non-default entry point *init*, which is used to initialize the mode before execution. Since agents never terminate, their top-level modes do not have non-default exit points.

An object-oriented feature of CHARON is that declarations of modes and agents act as classes. A parameterized declaration of a mode or an agent can be instantiated in a model multiple times with different values of parameters.

Semantics. CHARON is given formal compositional trace semantics. Each agent or mode is characterized by its interface and the set of traces it allows. Traces of a mode are formed by the flows defined by the mode constraints, interleaved with discrete steps of the mode, in which a mode transition is taken, updating local and output variables, and discrete *environment steps* that change the values of input variables. The set of traces of a composite mode can be computed from the traces of the submodes. While executing in one of the submodes, the mode follows a trace of the active submode that complies with the constraints of the mode.

A primitive agents has as its traces the traces of its top-level mode, restricted to the global variables of the agent. A trace of a composite agent is such that, when projected on the global variables of a sub-agent, it yields a trace of the sub-agent. Semantics of agents is also compositional. The set of traces of an agent can be computed from the sets of traces of the sub-agents.

A motivational example. We use a simple example throughout the paper to illustrate the facilities of CHARON. It represents a swimming pool equipped with a pump that controls the water level, and a sign that tells whether the water is deep enough to swim. The architecture of the model is shown in Figure 1. It consists of three agents, `Pool`, `Pump` and `Sign`. The first agent represents the water in the pool and its behavior is given by a single differential equation relating the flow of water and its level. Two other agents are instantiations of parameterized agents `WaterPump` and `Switch`. Their top-level modes are presented in Figure 2. The agent `WaterPump` controls the water flow. The pump can be turned on or off, maintaining constant flow: when the pump is on, water flows into the pool, when it is off, the water flows out of the pool. Modes `On` and `Off` are instances of the mode `SteadyMode` with different values of parameters. In addition, the pump has two transient modes, `TurnOn` and `TurnOff`. These modes are instances of the mode `TransientMode`, when the water flow smoothly changes from one steady-mode level to the other. Entry transitions of the primitive modes in the example are trivial and we omit them in the figures. Initially, the pump starts in the `On` or `TurnOff` submode depending on the water level, as prescribed by the entry transitions. Then, the pump cycles through on and off phases.

3 Fundamentals of qualitative reasoning

3.1 Qualitative variables

A qualitative variable has an associated type, or *quantity space*. A quantity space consists of a finite set of *landmarks*. A landmark represents an "interesting" value of the variable and may be a symbolic or integer constant. We assume that landmarks of a variable are completely ordered. That is, when we consider a variable v with the quantity space $\{v_1, v_2, ..., v_n\}$, we will always assume $v_1 < v_2 < ... < v_n$.

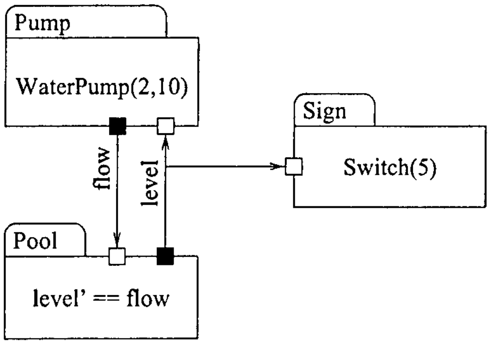


Figure 1: A swimming pool in CHARON

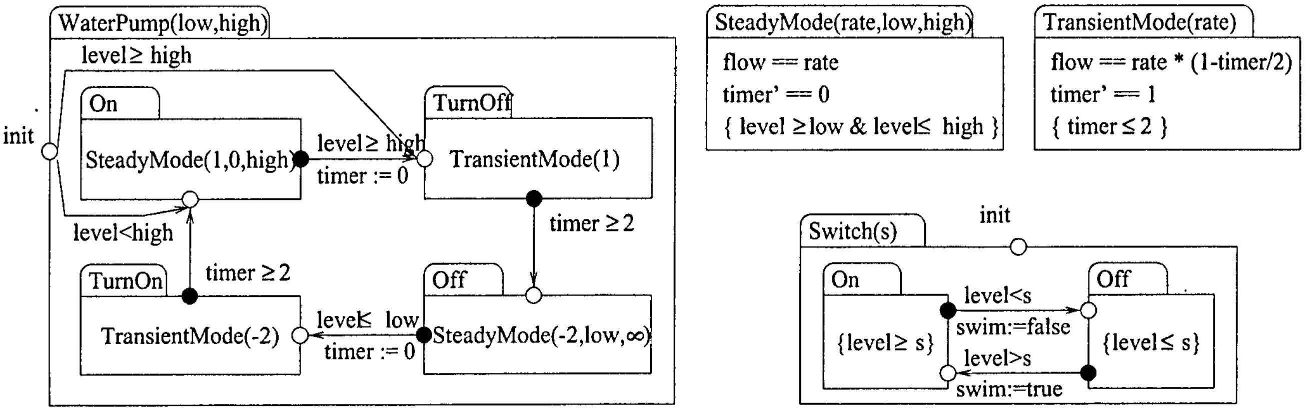


Figure 2: Declarations for the swimming pool

During an execution, we track not only the values of the qualitative variables, but also the directions of their change. It allows us to increase accuracy of qualitative behaviors. A *valuation* of a qualitative variable v is a pair (l, d), where l is a qualitative value of v and $d \in \{-, *, +\}$. The meaning of a valuation (l, d) for v is that the value of v is l and the first derivative with respect to time is negative if $d = -$, is zero if $d = *$, and is positive if $d = +$. For discrete qualitative variables the only possible value of d is their valuations is $*$.

During an execution, an increasing qualitative variable may either reach the next larger value or stop increasing. Similarly, a decreasing variable may reach the next smaller value or stop decreasing. A stationary variable may turn into either increasing or a decreasing one without changing its qualitative value. When the variable assumes its smallest landmark value, it cannot be decreasing; similarly, when it assumes the largest landmark value, it cannot be increasing. As an example, consider the possible evolutions of a single unconstrained qualitative variable with the quantity space $\{l_1, l_2, l_3\}$, represented as a state machine in Figure 3.

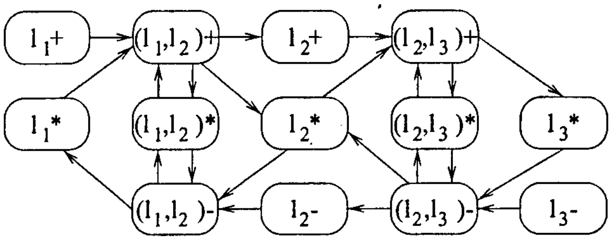


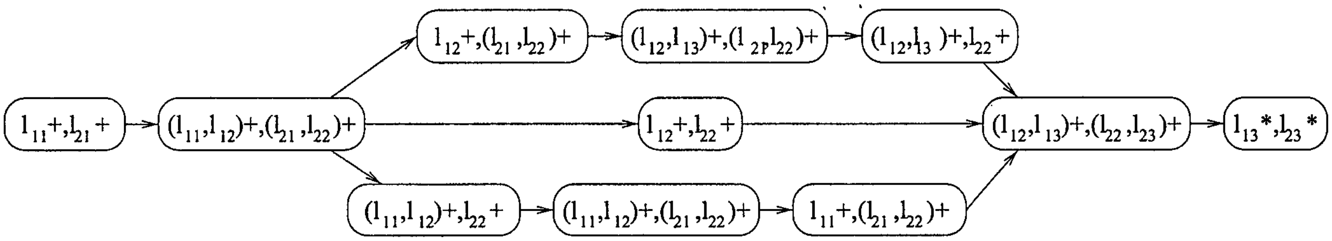
Figure 3: State machine for a single variable

3.2 Qualitative Constraints

Multiple variables in the same execution can evolve independently of each other. If variables are to evolve in a coordinated fashion, we need to introduce constraints.

Constraints over qualitative variables take the form $v \circ E_q$, where $\circ \in \{<, \leq, =, \neq, >, \geq\}$ and E_q is a *qualitative expression*. Qualitative expressions are constructed from qualitative variables and constants by means of qualitative operators described below. The quantity spaces of the left-hand and right-hand sides of a constraint must be the same.

Functional operators. A functional operator represents an underspecified function from a tuple of quantity spaces into a quantity space. An example of a functional operator is a monotonic function $M^+(v)$ for a qualitative variable v. A constraint of the form $v_1 = M^+(v_2)$ specifies that in every state of the execution, the directions of change in the valuations of v_1 and v_2 agree. In addition, the relation between some elements of the quantity spaces of v_1 and v_2 may be known. In this case, the valuations also have to agree on those elements. For example, consider a model with variables v_1 with the quantity space $\{l_{11}, l_{12}, l_{13}\}$ and v_2 with the quantity space $\{l_{21}, l_{22}, l_{23}\}$. Let the constraint be $v_1 = M^+(v_2)$ with the set of related values $\{(l_{11}, l_{21}), (l_{13}, l_{23})\}$. The "increasing" part of the state machine shown below. Symmetric "stable" and "decreasing" parts are omitted.


$l_{11}+, l_{21}+$
$(l_{11}, l_{12})+, (l_{21}, l_{22})+$
$l_{12}+, (l_{21}, l_{22})+$
$(l_{12}, l_{13})+, (l_{21}, l_{22})+$
$(l_{12}, l_{13})+, l_{22}+$
$l_{12}+, l_{22}+$
$(l_{11}, l_{12})+, l_{22}+$
$(l_{11}, l_{12})+, (l_{21}, l_{22})+$
$l_{11}+, (l_{21}, l_{22})+$
$(l_{12}, l_{13})+, (l_{22}, l_{23})+$
$l_{13}*, l_{23}*$

Arithmetic qualitative operators. Arithmetic qualitative operators are special cases of functional operators. An arithmetic qualitative operator is a mapping from a pair of quantity spaces to a quantity space. When components of the quantity spaces are integer constants, the natural rules can be used to define the arithmetic operators. For symbolic constants, the user must specify the mapping explicitly. For example, an addition operator from $\{0, \mathit{On}, \mathit{Inf}\}^2$ to $\{0, \mathit{On1}, \mathit{On2}, \mathit{Inf}\}$ may be given as $\{((\mathit{On}, 0), \mathit{On1}), ((0, \mathit{On}), \mathit{On1}), ((\mathit{On}, \mathit{On}), \mathit{On2})\}$. Addition and multiplication operators are always commutative and monotonic in both arguments; the landmark value 0 is always used naturally in all arithmetic operators. If quantity spaces are signed (that is, 0 is an element of the quantity space), multiplication of positive values yields a positive value, etc. Other properties of the arithmetic operators (such as associativity) may not be satisfied.

Qualitative differential constraints. In addition to constraints on variables, qualitative constraints can apply to first derivatives of variables. Remember that a valuation of a qualitative variable includes a three-valued component representing the direction of its change. A qualitative differential constraint constrains this component of the valuation. Values of the right-hand side expression are taken in relation to 0, which must be contained in the quantity space of the expression. For example, constraint $v' = 1$ means that the direction-of-change component of the valuation for v is always +; that is, v monotonically increases.

4 Qualitative CHARON

In this section, we describe QCHARON, that replaces real variables of CHARON with qualitative variables. The change affects only the modes. The agent hierarchy and interfaces of agents are unaffected (except that types of agent variables change to qualitative types).

We introduce the following quantity spaces for the variables used in the swimming pool example:

level	{ 0, *Lo*, *Swim*, *Hi*, *Owf* }
flow	{ *mInf*, *Out*, 0, *In*, *Inf* }
timer	{ 0, *Ready*, *Inf* }

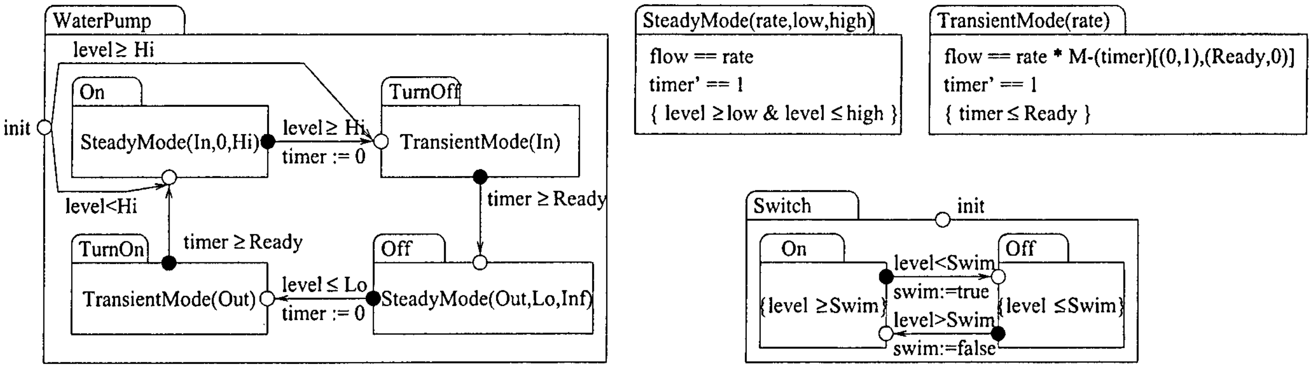


Figure 4: Declarations of qualitative agents and modes

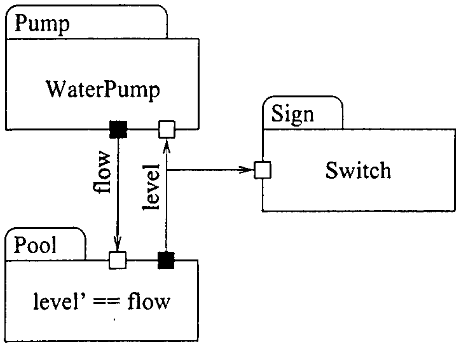


Figure 5: Architecture of the qualitative model

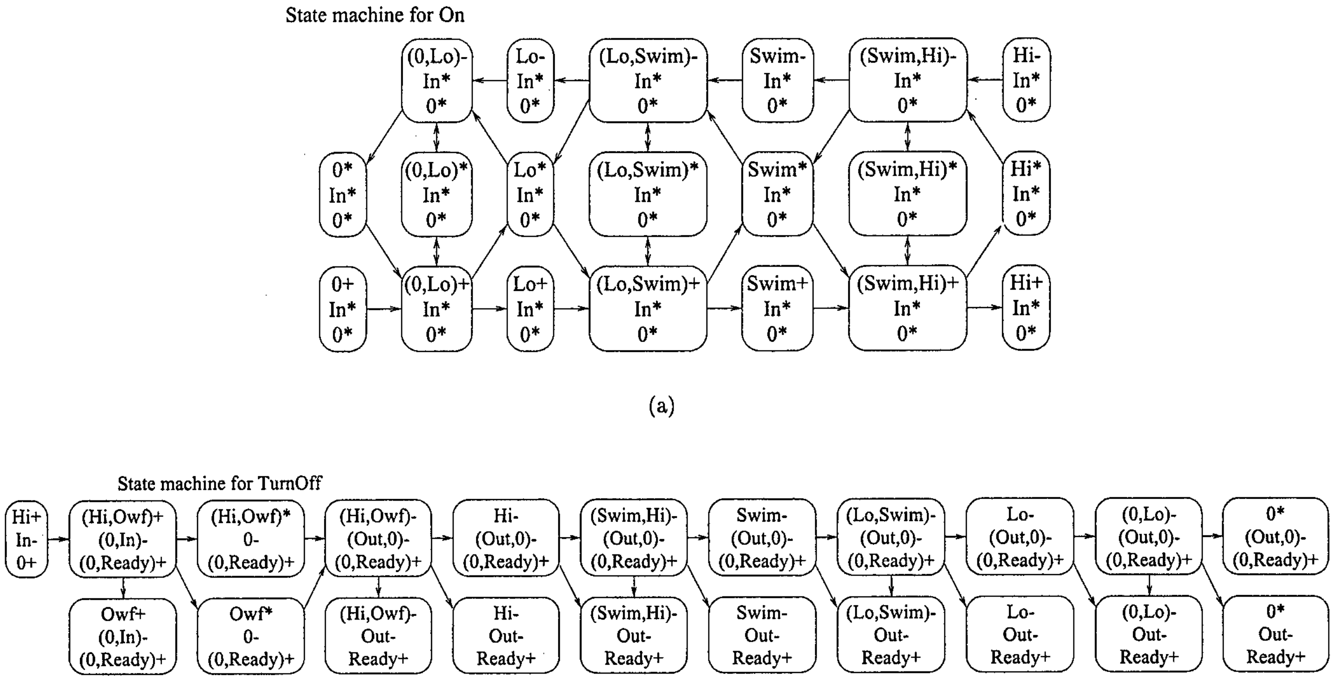
The landmarks for each variable are as follows: water level is at *Lo* when the pump has to be turned on, at level *High* the pump needs to be turned off, and it is safe to swim when the level is above *Swim.* When the flow of water is enough to make water level rise, the value of the flow is *In*, if the level is decreasing, the value is *Out.* The value 0 means that the level is constant. These landmark values for flow are chosen to be used in the differential constraint of the Pool agent. Finally, the timer has only one interesting value: duration of the interval that the pump spends in a transient state, denoted *Ready.* Note that all these values are either constants or parameters in the CHARON model.

We also need to turn the expressions of the CHARON model into qualitative expressions. Consider the differential equation of mode TransientMode, expression $rate * (1 - timer/2)$ becomes $rate * (M^{-}(timer)[(0,1)(Ready,0)])$. It represents a monotonically decreasing function of *timer*, which has value 1 when *timer* is 0, and 0 when *timer* has the qualitative value *Ready.* We do not need to specially define the multiplication operator in this expression, because we are interested only in the sign of the expression. All other expressions in the example are transformed into the qualitative form by replacing concrete constants and parameters with qualitative constants.

Figure 4 shows the qualitative version of the swimming pool example. Figure 5, which represents the architecture of the qualitative model is the same as Figure 1 with the parameters removed. The agents WaterPump and Switch are no longer parameterized, because their parameters are used in a qualitative way, and are now captured as types of qualitative variables. However, not all parameters in the model are removed. Submodes of WaterPump are still parameterized, since they are instantiated multiple times with different values of parameters.

Semantics. Following the setup of CHARON, semantics of a QCHARON specification is given by the interface of the mode (its control points and global variables), and set of traces that the specification can produce. We will define set of traces in a bottom-up fashion, starting from the leaves of a behavioral hierarchy. For each mode, we will first capture the set of its executions as a state machine. It is important to notice that this state machine is a semantic object and does not have to be constructed explicitly during analysis of a QCHARON specification. Executions are projected onto the global variables of the mode to yield the set of traces of the mode.

First, a state of a mode with variables $v_1, ..., v_n$ is a tuple of valuations for the variables of the mode. A mode



(a)

(b)

Figure 6: Runs of the submodes of `WaterPump`

variable can be updated either in a discrete or in a continuous fashion. Discrete variables are changed by the transitions of the mode. Discrete variables can assume only landmark values during an execution. A continuous variable follows a flow, i.e. a differentiable function, during an execution and thus can assume as values its landmarks and the intervals between adjacent landmarks.

Executions of a primitive mode are represented as a state machine, where each state corresponds to a state of the mode. Transitions represent possible changes in variable valuations, such that valuations in the states connected by a transition agree with all the constraints of the mode. A *run* of a state machine is a sequence of states such that every two consecutive states in a sequence are connected by a transition. To represent executions, we extend the state machine with special nodes that do not correspond to a state of the mode, but capture entry and exit points of the mode. Each entry point is connected by a transition to every state in which the values of variables agree with the guard and the action of the entry transition attached to the entry point. Every state is connected by a transition to the exit point node, since an execution can be interrupted at any time.

We show the state machine for `SteadyMode` (instantiated as `On`) in Figure 6(a). In the mode `On`, variables *flow* and *timer* are constant and variable *level* is an input variable whose value is constrained by the invariant of the mode and the direction of change is unconstrained. Figure 6(b) shows a fragment of the state machine for `TransientMode` (instantiated as `TurnOff`). In this mode, *timer* increases, *flow* decreases, and *level* is unconstrained. The fragment is chosen to comply with the constraint on level from the agent `Pool`, which will be applied when behaviors of individual components are composed into behaviors of the whole system. Since the entry transitions of the modes are trivial, the entry node of each state machine is connected to every state and we do not show them to avoid cluttering the figure.

We can now give semantics to composite modes by first extending the mode hierarchy at the leaves, replacing the primitive modes with their respective state machines. The state machine representing the executions of a composite mode m is obtained by "flattening" this hierarchical state machine into an ordinary state machine. To construct the flattened state machine of a mode from the flattened state machines of submodes, we perform the following steps.

1. Connect by transitions the states in the state machines of the submodes according to the transitions of the mode. Consider a mode m with submodes m_1 and m_2. Let m_1 have an exit point x and m_2 have an entry

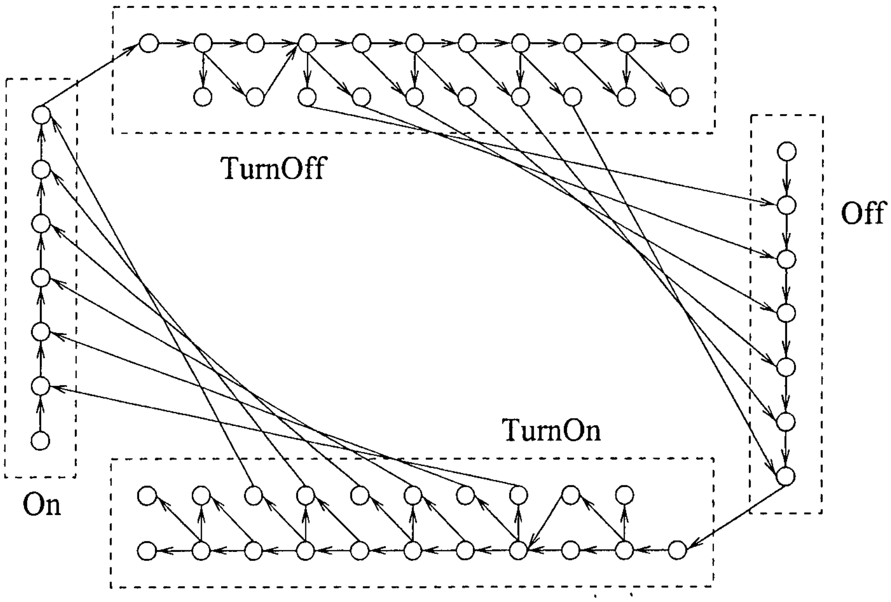


Figure 7: Qualitative runs of the swimming pool model

point e. Let t be a transition of m from x to e. Consider a state s_1 in the state machine of m_1 that is connected by a transition to the node corresponding to x, such that s_1 satisfies the guard of t. The state s_1 is connected by a transition to every state s_2 that agrees with the action of t. This operation is repeated for every s_1 in m_1 and for every t in m.

2. Next, we introduce the nodes for the control points of the mode and introduce transitions similarly way to the regular transitions: for each entry transition from an entry point e to an entry point e_1 of a submode m_1, we add a transition from the node for e to every state in the state machine of m_1 that is connected to e_1 and agrees with the guard and action of the entry transition.
3. After all transitions have been introduced, the nodes for the submode control points are removed.

Behaviors of a primitive agent are the same as the behaviors of its top-level mode. For a composite agent, we can compute the flattened state machine by taking a product of the state machines for the sub-agents, in which a transition is possible if and only if it is allowed by constraints in all the agents. In the swimming pool example, when we compose agents `WaterPump` and `Pool`, traces of `WaterPump` now have to satisfy the relationship between the variables *flow* and *level* prescribed by the `Pool`. In particular, this restriction effectively reduces the state machine of Figure 6(a) to the bottom row of states. We show the flattened state machine for the swimming pool example in Figure 7. To avoid cluttering the figure, we omit the labels of the states, but group together the states corresponding to executions within the same submode of `WaterPump`.

In the same way as CHARON, the semantics of QCHARON is compositional, making the construction of the flattened state machine unnecessary. The set of traces permitted by the state machine of a mode can be computed from the transitions and constraints of the mode and the sets of traces of the submodes.

5 Conclusions and Discussion

We have presented preliminary results on the construction of conservative approximations of CHARON specifications by means of qualitative reasoning. The approach differs both from the existing abstraction techniques for hybrid systems analysis and from traditional uses of qualitative reasoning. A lot remains to be done to turn this approach into an abstraction methodology for hybrid systems, but the first impression is encouraging.

Comparing our approach with that of qualitative simulation [12], we note that the hierarchical state machine yields a much more compact representation of the set of execution traces, in general, than an explicit representation. For comparison, we modeled our swimming pool example in QSIM, the foremost tool for qualitative simulation.

Much to our surprise, the problem turned out to be intractable for QSIM. It exceeded the limit of 500 traces that we set for the simulation, and ran out of memory with the trace limit removed.

Comparing the proposed technique to the abstraction techniques for hybrid systems described in [3], it is clear that qualitative reasoning yields coarser abstractions than other existing techniques. This has its advantages and disadvantages. On the one hand, qualitative abstractions are much easier to compute and manipulate. This allows us to handle larger specifications. On the other hand, qualitative abstract models are much less precise than state-of-the-art techniques. They admit many behaviors that the original system cannot exhibit. In the discussion below, we consider ways to improve precision of the abstraction without incurring too much overhead.

Our future work on this topic will concentrate on the following aspects:

- **Improving accuracy of abstractions.** A qualitative description of a mode or an agent represents all possible values parameters and constants in the model, because they are now assume qualitative values. This makes it more difficult to check properties of concrete systems. In effect, the question "does model A have property B?" in qualitative analysis becomes "can we select the values for parameters and constants in A such that the resulting model has property B?" As a result, a qualitative model will allow more qualitative behaviors than the original hybrid model, instantiated with a fixed set of parameters, would.

  In terms of our swimming pool example, the question whether the pool can overflow is answered positively. Indeed, if we choose the value of `High` too close to `Owf`, an overflow is possible in the `TurnOff` mode, while the water level is still rising above `High`. At the same time, parameters in the original swimming pool example were chosen so that overflow cannot occur. In order to get a more precise answer, we need to constrain the model further to express the relative values of qualitative landmarks. The problem can be addressed from two directions. On the one hand, *semi-quantitative* reasoning [4] extends purely qualitative reasoning with partial numerical information. The second approach involves model refinement techniques such as proposed in [6]. These two approaches will be the main direction of our future research in this area.

- **Local landmarks.** We observe that not every landmark value of a variable needs to be considered in every mode. For example, the value *Swim* of the variable *level* is used only in the agent `Switch`, and values *Low* and *High* are used only in the agent `WaterPump`. We can use this fact to reduce the sizes of mode state machines, and refine them as needed during analysis.

- **More complex functions.** Functions used in our example are very simple, which makes the qualitative abstraction easy to perform. In general, providing accurate qualitative representations for a complex function may be difficult. The problem has been studied in the context of qualitative simulation previously [8]. We will explore *mode splitting* to make construction of qualitative representations simpler. With this technique, we partition the ranges of variables in a CHARON model in such a way that in each block of the partition the function can be simplified or approximated differently, yielding expressions with simpler qualitative form in each case. Then, we introduce a separate mode for each block of the partition, with additional invariants to ensure that the values of variables are within the block. Transitions between the new modes will correspond to the execution moving from one block of the partition to another. In this way, a mode in a CHARON model will correspond to a number of modes in QCHARON, but each mode will provide a more precise approximation.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Proceedings of Hybrid Systems: Computation and Control, Third International Workshop*, volume 1790 of *LNCS*, pages 6–19. Springer-Verlag, 2000.

[2] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control, Fourth International Workshop*, March 2001.

[3] R. Alur, T.A. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 2000.

[4] D. Berleant and B. Kuipers. Qualitative and quantitative simulation: Bridging the gap. *Artificial Intelligence Journal*, 95(2):215–255, 1997.

[5] G. Brajnik and D.J. Clancy. Control of hybrid systems using qualitative simulation. In *Working notes from the 11th International Workshop on Qualitative Reasoning about Physical Systems (QR-97)*, June 1997.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV '00*, July 2000.

[7] J. De Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[8] A. Farquhar and G. Brajnik. A semi-quantitative physics compiler. In *Working Papers of the International Workshop on Qualitative Reasoning (QR-94)*, 1994.

[9] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[10] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of CAV '97*, pages 72–83. Springer-Verlag, July 1997. introduced predicate abstraction.

[11] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. *Journal of Computer and System Sciences*, 57:94–124, 1998.

[12] B. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge.* MIT Press, 1994.

[13] T. Loeser, Y. Iwasaki, and R. Fikes. Safety verification proofs for physical systems. In *12th International Workshop on Qualitative Reasoning*, pages 88–95. AAAI Press, May 1998.

[14] B. Schults and B. Kuipers. Proving properties of continuous systems: Qualitative simulation and temporal logic. *AI Journal*, 92:91–129, 1997.

Author Index